A DEDUCTIVE FAULT SIMULATOR FOR LOGIC CIRCUITS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

Major B. K. BHATIA

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
JULY, 1976

DEDICATED to -

The Corps of Electrical
and
Mechanical Engineers
(INDIAN ARMY)

CERTIFICATE

CERTIFIED that the work, 'A DEDUCTIVE FAULT SIMULATOR FOR LOGIC CIRCUITS' has been done by Major B.K. Bhatia under my supervision and that it has not been submitted elsewhere for a degree.

K.P.R. Prabhu

Dr. K.P.R. Prabhu
Visiting Research Associate
Department of Electrical Engineering
and Computer Science
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

IIT Kanpur
July 1976

ACKNOWLEDGEMENT

The author feels indebted to Dr. K.P.R. Prabhu for his constant guidance and encouragement in the accomplishment of a task which needed deep analytic study and colossal programming effort. With all his technical talent and practical know-how, Dr. Prabhu could always recommend 'easy-to-implement' solutions to all intricate and puzzling problems.

The topic of 'Fault-Simulation', was, however, introduced first by Dr. K.L. Kodandapani who had cultivated enough interest in the author to undertake this work. The stimulus and motivation provided by him is acknowledged with gratitude.

Amongst others, the names of Shri S. Kapoor and Sri R.P. Suri of Computer Centre, Indian Institute of Technology, Kanpur, need a special mention for their occassional help in debugging the simulator program in its development stages.

Finally, the author would like to thank Mr. H.K. Nathani for typing the manuscript.

- Major B.K. Bhatia

IIT Kanpur
July 1976

INDEX

ABSTRACT

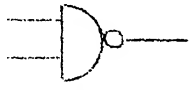
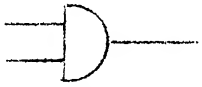
The need for accurate, flexible and efficient fault simulation has steadily increased during the recent years due to the increasing complexity and dimension of logic circuit design. Fault simulation is becoming a very important part of many design automation systems.

Various techniques exist for the development of these fault simulation systems. The DEDUCTIVE approach, compared to PARALLEL fault simulation methods, has a unique attribute in that the fault-free circuit and all the faults can be treated in one simulation pass. For larger circuits this method, in the literature, has been considered far more efficient although, due to the unpredictable nature of 'FAULT LISTS' behaviour, the requirement of efficient dynamic storage management is vital and is reported to present a serious problem (as well as challenge) to the implementor.

We have attempted to implement this technique for combinatorial logic circuits. Only the 'STUCK-AT' type of faults have been modelled and the accuracy possible with '3-VALUE' simulation using a 'UNIT TIME DELAY MODEL' has been considered. The dynamic storage allocation has been made using extensively the 'LINKED LIST DATA STRUCTURE' concept. Simulation efficiency has been increased by following the 'SELECTIVE TRACE' procedure and by designing algorithms for fast set operations. A lot of flexibility has been incorporated in the simulator by leaving enough scope for subsequent inclusion of higher levels of simulation capability (viz., ELEMENT and FUNCTIONAL levels), though we have restricted our work to the GATE level simulation only. It is felt that the extension of this model to include variable-time delay for different gates and multi-value simulation will be a useful exercise from the view-point of achieving better simulation accuracy.

In summary, this paper involves the development of a fault simulator in FORTRAN IV language for combinational logic circuits by implementing the deductive simulation technique. This is intended to be useful both to the logic circuit designer and to the logic circuit maintenance engineer.

LIST OF SYMBOLS/ABBREVIATIONS USED

	NAND Gate
	AND Gate
	NOR Gate
	OR Gate
	X-OR Gate
$\cup$	Set Union
$\cap$	Set Intersection
$\ominus$	Set Difference
+	Logical OR Operation
$\{ABC\}$	Logical AND of A,B and C
+	EXCLUSIVE-OR Operation
TTL	Transistor -Transistor Logic
DTL	Diode-Transistor Logic
MOS	Metal Oxide Semiconductor
CCT	Circuit
IC's	Integrated Circuits
$\bar{X}$	Not X (or BAR X)

CHAPTER 1

I N T R O D U C T I O N

CHAPTER 1

INTRODUCTION

1. OBJECTIVE

This project involves the development of a fault simulator in FORTRAN IV language for combinatorial logic circuits. The simulation technique adopted is ARMSTRONG's concept of 'FAULT LISTS' which is the key-tone of the 'DEDUCTIVE' approach. The program is machine-independent though its implementation has been tried out on IBM-7044 computer.

The main aim of this exercise is to help the logic circuit designer in the derivation of fault-detection test-sets, and in the logic verification of his circuit. Also fault-diagnostics on equipments using logic circuits can be made easy by simulating the circuits and by generating a fault dictionary for the same. This one-time record, the fault dictionary, shall facilitate maintenance and economize on time.

2. NECESSITY

As the level of logic circuit integration increases, it becomes more and more difficult to build 'bread board' models. An effective computer simulation of the logic circuit becomes vital under these circumstances. Also since the designer of the circuit feels interested in

the faulty behaviour of his circuit under different input conditions, FAULT SIMULATION becomes necessary.

Moreover, in the interest of speedy maintenance of an equipment using logic circuits, it is necessary to reduce the 'down-time' of the equipment to increase its 'period of availability' to the user. This can be achieved only by curtailing the time spent on 'fault-diagnostics'. To assess the faults by monitoring waveforms at various test-points in the circuit is a laborious task compared to examining the simulated output behaviour of the circuit for a given input combination. The problem becomes more pronounced as the size of the circuit increases. Knowledge, experience and reaction-time of operators being variable, a standardized solution of the fault-diagnostic problem becomes important. The requirement obviously is of a 'fault-simulator' of the 'universal' type, which can be used for all categories of logic circuits. It should even be possible to simulate on a large digital computer any other computer system or its sub-systems, and study the response of the simulated system to different input stimuli to assess the functional performance of the same. In-case of erroneous results, it should be feasible to assess the cause and pin-point the fault.

With the rapid advancement in digital IC's technology, various digital simulation techniques are gaining increasing importance. Concurrent simulation of fault-free circuits and the effect on the circuit of a 'small set' of single permanent faults is achieved by the 'Parallel Simulator'. The 'Deductive' simulator', on the other hand, simulates concurrently the fault-free circuit and the effect on the circuit of 'all' single permanent faults. The latter approach has been preferred for implementation for its novelty and uniqueness.

3. APPROACH AND SCOPE OF WORK

The basic logic device is a 'GATE'. This could be an AND, OR, NAND, NOR, XOR or an INVERTOR (NOT) gate. 'FLIP FLOPS' or 'GATES grouped with FLIP FLOPS' are termed as 'ELEMENTS'. The main difference between a 'gate' and an 'element' is that the latter may be multiple input and output devices, while gates are essentially a single output device. Also ordering of inputs and outputs for elements has a definite significance in evaluation, whereas the ordering of the gate inputs need not be considered in evaluation of the gate. A natural extension of an 'element' level simulator is a 'functional' simulator. The difference is only of generality. For example, at element level we may have a 4-input, 16-output decoder.

At the functional level we shall have only an N-input, 2^N output decoder where 'N' is specified by the user.

The above discussion is merely to define the scope of simulation work undertaken. Only 'GATE' level models have been considered for reasons given below:

(a) It is more general a model and can encompass both the functional and element level models by breaking them up into their basic constituents. For example, the flip-flops can be expanded to their gate-level equivalences.

(b) Implementation of the simulator is made simpler since only a restricted set of gates is required to be modelled.

(c) It is not always possible to have a purely 'element' or 'functional' level description of any logic circuit. Gates shall have to be simulated as intermediate connecting links thus affirming that 'gate' level simulation is quite indispensable.

The only probable draw-back of gate-level simulation is that when the user desires to simulate larger systems where gate level descriptions are neither desirable nor possible, then the simulator becomes ineffective. For that the scope for addition of element level subroutines has been left in the program and 'element' level models

can be easily included without compromising with the efficiency of the simulator.

The scope of implementation has been restricted to purely combinatorial logic, though an attempt has been made to discuss the methodology of approach while tackling the simulation of 'FLIP-FLOPS' with the deductive technique. With the addition of subroutines for 'flip-flops' this simulator can be easily generalized to cater for all types of logic circuits. This has been discussed in Chapter 4. Fault lists have been worked out for some of the flip-flops though generalization of the algorithm remains to be attempted.

The simulator caters for a circuit with maximum of 500 logic gates, this binding having been imposed by the consideration of available memory space in the host computer (viz., IBM-7044). 'Linked List' structures have been generally employed in lieu of 'matrix' form of storage and operation for better memory utilization.

To refresh the reader's knowledge on digital simulation, a brief review of the existing simulation techniques has been made in Chapter 2. In this chapter the 'Deductive technique' has been discussed at length since this concept has ultimately been utilized in developing the simulator of Chapter 3. Chapter 3 illustrates various developmental steps in the design of the

'Deductive Fault Simulator', and enumerates the facilities provided to the user. Finally in Chapter 4 the scope of future development has been outlined. Some work done on the simulation of sequential circuits is also included. A copy of the program has been attached as an Appendix along with the results obtained for a small representative circuit. These results have partly been manually attempted in Chapter 3 for one of the test vectors. This is for cross-check, verification and proper assimilation of the deductive technique by the user.

CHAPTER 2

REVIEW OF SIMULATION TECHNIQUES

CHAPTER 2

REVIEW OF SIMULATION TECHNIQUES

1. TERMINOLOGY USED

Before we proceed to review various simulation techniques, let us understand some of the terminology subsequently used.

(a) Logic Circuit: As interconnection of logic gates with discrete input values (generally 0 or 1) constitute a logic circuit. This may also be termed sometimes as 'OBJECT NETWORK'.

(b) Digital Logic Simulation: This implies the making of a realistic model of the digital system's logic. This model will provide, in the form of computer printout, the signal-vs-time behaviour of output of the logic circuit.

(c) Node: Every vertex of a directed graph whose branches are the interconnecting wires in a logic circuit is termed as a node. For our purposes all logic gates shall be considered as nodes.

(d) Test Vector: A set of inputs $\underline{X}_i (= x_1, x_2, \ldots, x_n)$ applied at the 'n' input terminals of a logic circuit, constitutes a test vector. It is that input combination which produces an 'incorrect' output when a fault is present. For purposes of simulation, this has to be

specified either by the user, or generated randomly.

(e) Primary Inputs: Any input to a gate which forms a part of the test vector applied is termed as the Primary Input.

(f) Primary Outputs: We shall define primary outputs as the outputs of these gates which have no fan-out. In case the circuit to be simulated has a global feed back, we have provided a facility to the user of the simulator to specify the desired output points in addition to those sensed by the computer from the circuit data. This is discussed in Section 7 of Chapter 3.

(g) Test Points: These are the user-specified nodes where output is required to be printed for logic verification of the circuit.

(h) Gate Delays: The time elapsed before the effect of an input change to a gate is transferred to its output is termed as gate-delay. It varies from one gate to another and is generally of the order of a few nano-seconds.

(j) Fan-in: The maximum number of permissible input connections to a gate define its fan-in capability.

(k) Fan-out: The maximum number of inputs of different gates which are allowed to be connected to the output terminal of a single gate 'G' define the

fan-out capability of that gate 'G'. Precisely, fan-out specifies the loading restrictions on the output of a gate.

(l) Logic Fault: A logic fault is the one that causes the intended logic function of a gate to be changed to some other logic function - e.g., an AND gate becoming an OR gate. 'Stuck-at-faults' are a special case of logic faults in which either an input or output of a node gets stuck at '1' or '0' (briefly written as s-a-1 or s-a-0).

(m) Fault Lists: A set of faults which can individually or collectively alter the true output value of a node constitute the fault-list of the node.

(n) Two, Three or Multivalued Simulation: Whenever the input signals can take on only '1' or '0' as the values, it is called 'two-value simulation'. A natural extension is the addition of a "DON'TKNOW' (i.e., X) condition for purposes of initialization. This makes the 'three-value simulation' model (i.e., 0,1 and X). Multivalued simulators use more than 4 values for the signals and are effective in the analysis of 'logical hazards' in a circuit.

(o) Zero, Unit or Assignable Delay Models: 'Zero' delay simulation models assume that the effect of change in signal value is propagated through all the intermedia

nodes to the primary output points without any delay. 'Unit' delay model assigns a uniform fixed time delay to all the gates in a circuit. 'Assignable' or 'Variable' delay simulators basically permit the assignment of a single nominal (average) delay to each gate type (e.g., each AND gate in a logic network may be assigned a delay of 20 time units, while each OR gate may have a delay of 15 time units, etc.). This model, compared to 'zero' or 'unit' delay models is very significant in achieving simulation precision.

(p) Selective Trace: This is a technique to reduce the simulation time. It is based on the observation that if the output of a gate does not change with the change in its input excitation, then the fan-out of that gate remains unaffected, and need not be evaluated again provided all other inputs to the fan-out elements also remains unchanged. In other words, a gate is simulated only when one or more of its inputs changes state.

(q) Global Feedback: A feedback to any node from the circuit termination point will be termed as global feedback.

(r) Redundant Connections: A connection is said to be redundant if it can be cut without altering the output function of a node.

(s) Race Condition: This is applicable to FLIP-FLOPS. For a LATCH formed by NAND gates, a race is declared when the outputs of both gates are simultaneously zero, or both outputs of the gates are scheduled to be changed at the present time.

(t) Oscillations: A true value oscillation occurs when the circuit state is unstable as a result of some input condition. An oscillation is declared if the simulator simulates an arbitrary number N of increments of simulation time, and the circuit has not stabilized.

(u) Hard Faults: These are those faults in the circuit which cause the true output value (0 or 1) of the node to be complemented.

(v) Star Faults: These are those faults in the circuit for which output value of a node is not predictable.

(w) Spikes: A spike is defined to be a signal of shorter duration than necessary to change the state of the element.

2. EXISTING FAULT SIMULATION TECHNIQUES

In this section we shall briefly review the existing fault simulation techniques to familiarise ourselves with the various approaches to the problem of 'Fault Simulation'. Basically these approaches can

be classified in either of the following categories:

(a) Parallel fault simulation.

(b) Deductive fault simulation.

A comparision of these techniques has been made in the last section of this chapter. For the present we restrict ourselves to the understanding of the following methods:

(a) Truth Table Method

Compare the truth table of the normal and the faulty circuits.

Let inputs to a combinatorial circuit be $x_1, x_2, \ldots x_n$ and the outputs be ... ... $z_1, z_2, \ldots z_m$

where $Z_i = f_i(x_1, x_2, \ldots, x_n)$: $i = 1, 2, \ldots, m$

For any set of faults, F, and any fault α in F,

Let

$$Z_i^{\alpha} = f_i^{\alpha}(x_1, x_2, \ldots, x_n)$$

be the value of i-th output when the fault is present.

Then, an input vector $\underline{X^j} = (x_1^j, x_2^j, \ldots, x_n^j)$ is a test for detecting the fault 'α' if and only if

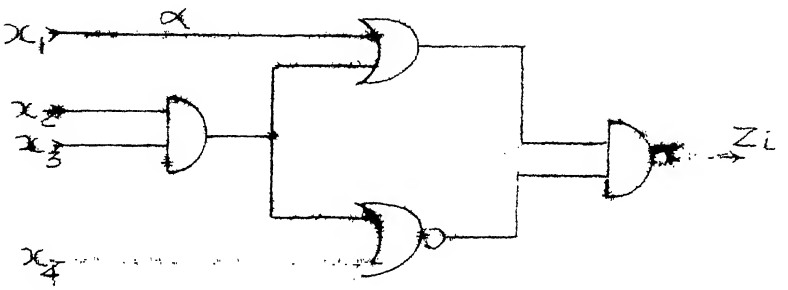
$$f_i(X^j) \oplus f_i^{\alpha}(x^j) = 1 \text{ for some } i, \ 1 \leqslant i \leqslant m$$

This test means the fault α is detected by applying the input $\underline{X^j}$ and observing the output Z_i.

Example

Consider the circuit shown below. Assume the wire α is s-a-1, and construct a truth table to detect this fault.

Circuit



Truth Table

Sl No	x_1	x_2	x_3	x_4	Normal O/P Z_i	Faulty O/P Z_i^α	$Z_i \oplus Z_i^\alpha$
1	0	0	0	0	0	1	1
2	0	0	0	1	0	0	0
3	0	0	1	0	0	1	1
4	0	0	1	1	0	0	0
5	0	1	0	0	0	1	1
6	0	1	0	1	0	0	0
7	0	1	1	0	0	0	0
8	0	1	1	1	0	0	0
9	1	0	0	0	1	1	0
10	1	0	0	1	0	0	0
11	1	0	1	0	1	1	0
12	1	0	1	1	0	0	0
13	1	1	0	0	1	1	0
14	1	1	0	1	0	0	0
15	1	1	1	0	0	0	0
16	1	1	1	1	0	0	0

The above table shows that only the test vectors serial Nos. 1,3, and 5 detect the fault 'α' s-a-1.

These test vectors are:

$$(x_1, x_2, x_3, x_4) = (0,0,0,0);(0,0,1,0);(0,1,0,0)$$

In the language of min-terms, this can be expressed as -

$$\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4$$

which is the Boolean expression for representing this fault.

In a similar manner, tests for detecting other faults in the circuit can be determined.

This method, as is evident, will be impractical even for circuits of moderate size because a truth table has to be constructed for every possible fault. In terms of computer time and storage, this technique appears to be uneconomical.

(b) Method of Boolean Differences

Boolean difference of a function $F(x_1, x_2, \ldots, x_n)$ with respect to one of its inputs x_i (denoted by $dF(x)/d_{xi}$) is defined as follows:

$$\frac{dF(x)}{dx_i} = F(x_1, x_2, \ldots, 0, \ldots, x_n) + F(x_1, x_2, \ldots, 1, \ldots,$$

[note - $\frac{dF(x)}{dx_i}$ is not a derivative but a symbol]

If $\frac{dF(x)}{dx_i} \equiv 0$, this means F is independent of x_i.

Similarly, if $\frac{dF(x)}{dx_i} \equiv 1$, any change in x_i will affect the output independent of the values of all x_j, $j \neq i$. In general, $\frac{dF(x)}{dx_i}$ will be a function of some (or all) of the x_j's, $j \neq i$.

The value of the function will depend on the value of x_i if and only if the remaining variables assume values such that $dF(x)/dx_i = 1$.

In order to test for a fault on x_i, we set x_i opposite to its faulty value and assign values to the remaining inputs such that $\frac{dF(x)}{dx_i} = 1$. The set of tests for a fault on x_i can be represented by the following expressions:

$$x_i \frac{dF(x)}{dx_i} \quad \text{for } x_i \text{ s-a-0}$$

and $$\bar{x}_i \frac{dF(x)}{dx_i} \quad \text{for } x_i \text{ s-a-1}$$

Various expressions to find the Boolean difference of complex circuits in terms of the Boolean difference of simpler circuits are given below:

(i) $$\frac{d\bar{F}(x)}{dx_i} = \frac{dF(x)}{dx_i}$$

(ii) $$\frac{dF(x)}{dx_i} = \frac{dF(x)}{d\bar{x}_i}$$

(iii) $$\frac{d}{dx_i}\left(\frac{dF(x)}{dx_j}\right) = \frac{d}{dx_j}\left(\frac{dF(x)}{dx_i}\right)$$

(iv) $$\frac{d[F(x)G(x)]}{dx_i} = F(x)\frac{dG(x)}{dx_i} + G(x)\frac{dF(x)}{dx_i} + \frac{dF(x)}{dx_i} \cdot \frac{dG(x)}{dx_i}$$

(v) $$\frac{d[F(x) + G(x)]}{dx_i} = \frac{dF(x)}{dx_i} + \frac{dG(x)}{dx_i}$$

$$(vi)\quad \frac{d[F(x) + G(x)]}{dx_i} = \bar{F}(x)\,\frac{dG(x)}{dx_i} + \bar{G}(x)\,\frac{dG(x)}{dx_i} + \frac{dF(x)}{dx_i}\cdot\frac{dG(x)}{dx_i}$$

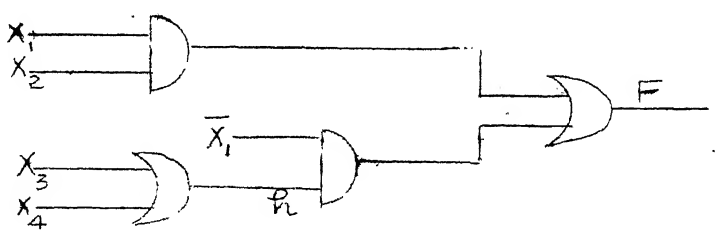
Tests for an internal wire 'h' in the circuit can be found by expressing F as a function of h, $F(x_1, x_2, \ldots, x_n, h)$, and h as a function of the inputs, $h(x_1, x_2, \ldots, x_n)$.

Then $h(x_1, x_2, \ldots, x_n) \cdot dF/dh$ gives tests for 'h' s-a-0 and $\bar{h}(x_1, x_2, \ldots, x_n) \cdot dF/dh$ gives tests for 'h' s-a-1

NOTE: The above results are neither being derived nor illustrated since these will not be relevant to our fault diagnostic technique. For details the reader may refer to [1].

Example

Consider the circuit shown below. Derive the tests for the wire 'h' s-a-0 and s-a-1.



$$F = X_1X_2 + \bar{X}_1(X_3+X_4) = X_1X_2 + \bar{X}_1h$$

$$= G(X_1, X_2) + H(x_1, h)$$

∴ By formula (vi) above

$$\frac{dF}{dh} = \bar{G}\,\frac{dH}{dh} \oplus \bar{H}\frac{dG}{dh} \oplus \frac{dG}{dh}\cdot\frac{dH}{dh}$$

$$= (\bar{x}_1+\bar{x}_2)\,\bar{X}_1 \oplus 0 \oplus 0 = \bar{\bar{X}}_1$$

Tests for 'h' s-a-0 are given by -

$$h\cdot\frac{dF}{dh} = (X_3 + X_4)\,\bar{X}_1 = \bar{X}_1X_3 + \bar{X}_1X_4$$

and for 'h' s-a-1 are given by -

$$\bar{h}\ dF/dh = (\overline{X_3+X_4})\,\bar{X}_1 = \bar{X}_1\bar{X}_3\bar{X}_4$$

Main limitation of this method is its applicability only to single output combinatorial circuits.

(c) Path Sensitizing Method:

Instead of using Boolean difference method to determine the conditions under which a change in the signal value on a wire in the circuit will affect the value of output, the conditions for propagating a change to an output along any path can be determined from a knowledge of the logic circuit. This is done by assigning input values to each gate along the chosen path such that its output depends on one particular input. The conditions required for a change in one of the inputs to a gate to cause a change in its output depend on the type of gate involved.

For AND and NAND gates all inputs except the changing one should be '1', and for OR and NOR gates these should be '0'. In general, all inputs, except the changing one, should bear the 'non-dominant' value for that gate.

Procedure for deriving tests using this method:

(i) Faulty wire is assigned a value opposite to the fault condition (i.e., '1' for s-a-0 fault and '0' for s-a-1 fault).

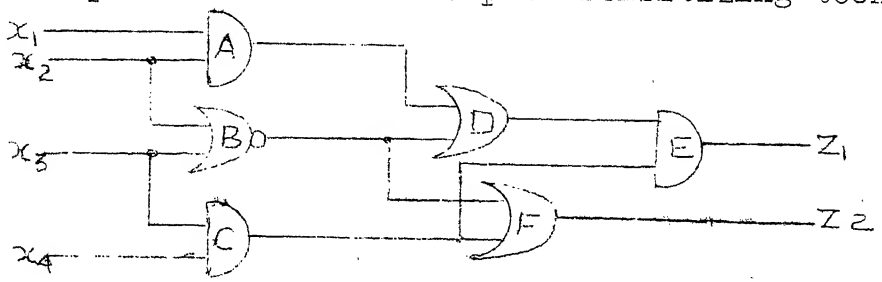
(ii) Choose a path from the fault to one of the output terminals.

(iii) Inputs to the gates along this path are assigned values so as to propagate any change on the faculty wire along the chosen path to the output terminal. The path is now said to be sensitized.

(iv) Trace back from the gates along the sensitized path towards the network inputs, and assign values to a sufficient number of inputs to obtain the desired signal values in the circuit. This procedure may not yield a unique set of inputs for sensitizing a particular path. An arbitrary choice is made wherever different possibilities exist.

Example

Refer to the circuit diagram shown below and explain the concepts involved in the path sensitizing technique -



Let the out-out of NOR gate 'B' be s-a-0.

(i) to detect s-a-0 fault, we require output of B = 1

(ii) Choose to sensitize the path BDE to output Z_1

(iii) i.e., for Z_1 to be 1, A = 0 and C = 1 is one of the combinations.

But B = 1 implies $x_2 = x_3 = 0$

This also makes A = 0 which we needed.

(iv) But in order to make C = 1, $x_3 = x_4 = 1$. This contradicts the value assigned to x_3 previously.

(v) If we start with A=1 and C=1, then x_2 has to be necessarily = 1 which does not make B = 1.

(vi) Since these are the only two possible combinations of values for A and C and BDE is the only path from B to Z_1, it follows that the fault 'B' s-a-0 cannot be detected at the output Z_1.

However, this fault can be detected at Z_2 since $x_2 = x_3 = 0$ sensitizes the path BF and also makes B = 1. Inputs x_1 and x_4 remain unspecified for this test, meaning thereby that all input combinations with $x_2 = x_3 = 0$ will detect this fault at Z_2.

The main draw-back of this method is that it sensitizes only one path in the circuit at a time. Hence the procedure is inadequate.

However, the concept of fault propagation explained in this method will be utilized by us in the DEDUCITIVE TECHNIQUE and it should be clearly understood.

(d) D-Algorithm Method

In this method it is important to identify two special types of inputs to a logic block given below:

(i) First type are those inputs which cause the output of the block (assuming single output blocks) to be different from its normal value, if a given fault is present in the block. These faults are represented by the 'Primitive D-cubes of the fault'.

(ii) Second type of inputs, represented by the 'Propagation D-cubes of a block' are those that cause the output of the block to depend on one or more of its specified inputs (and hence to propagate a fault on these inputs to the output).

We shall not go into various definitions involved, nor shall we attempt to discuss the methodology of this technique since these concepts are not utilized by us subsequently, but we explain with the example below the meaning of 'Primitive' and 'Propagation' D-cubes. (For details refer to [1].

Example

For a simple block (say NOR gate), and simple faults like an input s-a-1, both types of D-cubes can be written down by inspection

Say, if the input lead 'a' is s-a-1,

a = 0, b = 0 will cause the output c = 0 if the the fault is present, and c = 1 otherwise.

This is represented by the following primitive D-cubes of the fault

$\frac{a}{0}$ $\frac{b}{0}$ $\frac{c}{D}$ where 'D' represents the condition under which the normal output is '1' and the faulty output is '0'. (Note: This choice is arbitrary and opposite choice could have been made.)

On the other hand, if we are interested in propagating the effect of a fault (external to the particular gate) through the NOR gate, the following D

cubes of the block are of interest, assuming that only one input to the gate may be faulty

a	b	c
D	0	$\bar{D}$
0	D	$\bar{D}$

Here the interpretation of the symbol D is slightly different. D may be '0' or '1', but all D's in a D-cube always have the same value ($\bar{D}$ is the complement of D).

The drawback of this method is its inability to simulate all the single faults in the circuit simultaneously.

(e) Armstrong's Simulation Technique

Armstrong makes use of a procedure similar to path-sensitizing. For any given test, the normal machine is simulated and the results are used for determining the set of faults detected by it.

This method we have discussed in detail in the following section.

3. FAULT SIMULATION BY ARMSTRONG'S DEDUCTIVE METHOD

Discussed below is the concept of Deductive technique as projected by Armstrong with some additions and modifications as suggested by various other authors on this subject (Refer [3]).

In this simulation approach, Douglas B. Armstrong, (Refer 1 - 3) has termed all basic logic elements (i.e., gates and Set/Reset Flip Flops) as NODES. A gate is represented by one node, while a flip-flop has two nodes (one each for its input terminals).

(a) Levelling of Nodes: Various nodes can be grouped under different logic levels as classified below -

(i) First level nodes - These are the nodes having atleast one of their input terminals connected to primary input points.

(ii) Second level nodes - Atleast one of their inputs is connected to the output of the first level nodes.

(iii) n-th level nodes - In general, n-th level nodes are those having atleast one of their inputs connected to the output of (n-1)th level nodes. In case another input of the same node is connected to the output of a (m-1)-th level node, then this node also belongs to the m-th level. This indicates that any single node can belong to several levels depending on its FANIN connections.

(b) Simulator Structure: The simulator is table-driven. The circuit description is contained in tables which are accessed by the program during execution. Simulation is at gate level though several gates may sometimes be grouped into a single logical unit (called element). The simulated program updates its computation at successive fixed intervals of simulated time equal to the average delay per gate. The variation in delays is considered to be too small to affect the results of

simulation and no race analysis are undertaken. Feedback loops in the circuit are not identified and do not affect the program. This is because of the assumption of fixed delays associated with every gate. The 'selective trace technique' is used to avoid recomputation of nodes with unchanging input vectors. In other words, the output of any node is computed if and only if one or more of its input values changed during the preceding simulation interval.

(c) Simulator Operation: In the simulator, the identification of nodes by levels is provided automatically by means of a linked-list structure which describes the circuit topology. No formal levelling of nodes is necessary. At the beginning of simulation the linked list circuit description is stored in the host computer. Then the application of test vector to the circuit input terminals is simulated. The true-value outputs of the first-level nodes are then computed in response to the applied test vector.

Whenever the output of a node is computed, the list of single faults which will cause the output of the node to be different from its normal value is also determined. This is called the fault list of the node.

Likewise the simulator computes fault lists associated with the output terminal of each first level node. The faults in a particular list are the ones arising within the associated nodes, and which are detectable at the nodes output terminals when in its current logic state. That is, each fault in a list, if inserted singly in the circuit, would cause complementation of the true output of the associated node.

Next the true logic states and the associated fault lists for the second-level nodes are computed. In this case, and for all succeeding levels of nodes, the faults in a particular list arise from two sources discussed below:

(i) the faults arising from within the associated node which are detectable at its output when in the current logic state.

(ii) the faults which propagate to the input terminals of a node from the previous level nodes, and are transferrable to the output of this node in the current logic state.

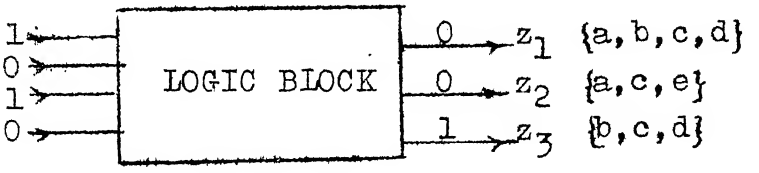
The above computations are repeated, level by level, throughout the circuit. In this manner a fault-list is generated for each node output and is updated, as

necessary, for every change in the input vector. These fault lists provide a dynamic record of all faults that are detectable at the output of the associated node at the current simulation time, during execution of a sequence of tests.

Error symptoms, consisting of signal values on monitored outputs in the presence of faults, are determined from the fault lists of these outputs as shown in the following example:

Example

The figure below shows true value outputs and associated fault lists at the primary output points of a logic block. Find the 'fault/error symptoms (and) correlation table' for the input combination specified:

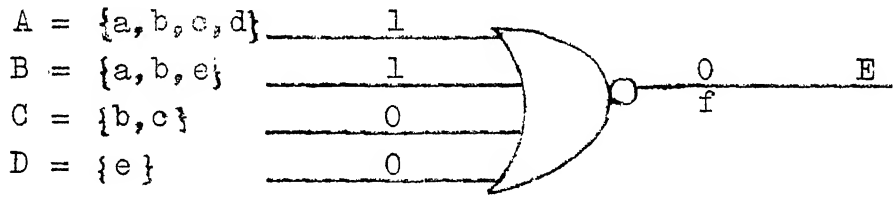


Since fault 'a' appears at both the outputs z_1 and z_2, its error symptom will be 1 1 1 ; i.e., the output value 1 1 1 at Z_1, Z_2, z_3 respectively will indicate the presence of fault 'a' in the logic circuit. Like-wise other error symptoms can be determined and tabulated as shown below:

Faults	Error-Symptoms		
	z_1	z_2	z_3
a	1	1	1
b,d	1	0	0
c	1	1	0
e	1	0	0

Such a tabulation which identifies the physical location of a fault can be termed as 'FAULT DICTIONARY'.

(d) Fault List Evaluation Technique: Let us first demonstrate the derivation of the fault-list for a NOR gate shown below:



(i) The fault lists associated with the input terminals of the NOR gate are specified by A,B,C,D, respectively with specific faults listed as shown therein.

(ii) The test vector is 1100

(iii) The true output of the gate can be evaluated and is equal to '0'.

(iv) We require to compute the fault list 'E' associated with the output.

It is observed that several faults appear in more than one fault list. This indicates re-convergent fan-out which means that the effect of these faults will propagate to the NOR gate under consideration along

different paths, when the particular input combination is applied to the circuit.

In order for the effect of a fault to propogate through the NOR gate, the output must change to '1', and hence all inputs must be '0'. This is possible only for faults that are contained in both 'A' and 'B', but not 'C' or 'D'. If the fault causing the output of the nor gate to be s-a-1 is denoted by 'f', we have:

$$E = A \cap B \cap \bar{C} \cap \bar{D} \cup \{f\}$$

$$= A \cap B \cap (\overline{C \cup D}) \cup \{f\}$$

$$= \{a,b,c,d\} \cap \{a,b,e\} \cap (\overline{\{b,c\} \cup \{e\}}) \cup \{f\}$$

$$= \{a,b\} \cap \{a\ d\} \cup \{f\}$$

$$= \{a,f\}$$

which is the fault list at the output.

In the same example, if normal inputs to NOR gate are 0010, and the input fault lists are unchanged, the fault list at the output

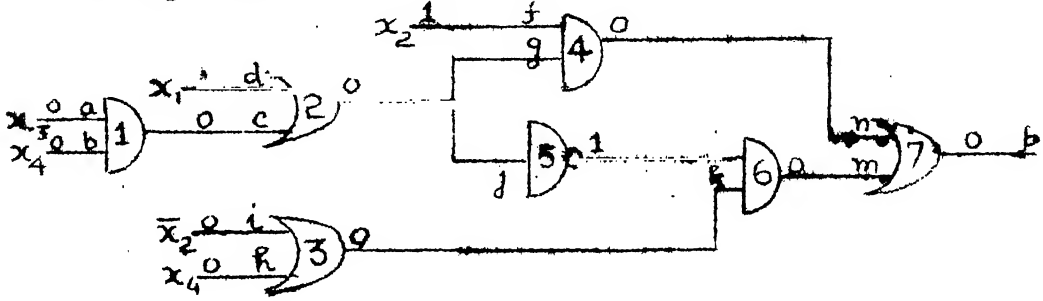
$$E = \{f\} \qquad \text{VERIFY!}$$

Fault lists for other types of gates can be derived in a similar manner as illustrated in the above example.

But let us try to evaluate fault list at the output of following circuit with multiple nodes so as to acquire a better understanding of the concepts involved.

Example

Figure below shows the normal signals with the test $\bar{x}_1\ x_2\ \bar{x}_3\ \bar{x}_4$ applied. Denoting s-a-0 and s-a-1 faults by subscripts 0 and 1 respectively and the fault list of each pin by the corresponding capital letters, we obtain the following fault lists for the circuit:



The true value at the output of each node has been specified. Fault lists are as under:

$A = \{a_1\}$

$B = \{b_1\}$

$C = \{A \cap B\} \cup \{C_1\} = \{C_1\}$

$D = \{d_1\}$

$E = \{D \cup C\} \cup \{e_1\} = \{c_1, d_1, e_1\}$

$F = \{f_0\}$

$G = E \cup \{g_1\} = \{c_1, d_1, e_1, g_1\}$

$H = \{h_1\}$

$I = \{i_1\}$

$$J = E \cup \{j_1\} = \{c_1, d_1, e_1, j_1\}$$

$$K = I \cup H \cup \{k_1\} = \{i_1, h_1, k_1\}$$

$$L = J \cup \{l_0\} = \{c_1, d_1, e_1, j_1, l_0\}$$

$$M = (K \cap \bar{L}) \cup \{m_1\} = \{i_1, h_1, k_1, m_1\}$$

$$N = (G \cap \bar{F}) \cup \{n_1\} = \{c_1, d_1, e_1, g_1, n_1\}$$

$$P = N \cup M \cup \{p_1\} = \{c_1, d_1, e_1, g_1, n_1, i_1, h_1, k_1, m_1, p_1\}$$

It is observed that for this particular input combination many results can be detected at the Primary output point P.

Let us understand how the fault list 'M' has been worked out as -

$$M = (K \cap \bar{L}) \cup m_1$$

at the output of node No. 6.

Since the true output value is '0', any fault that can change it to '1' qualifies to be included in list M. These faults are as follows:

(i) output struck at '1' i.e., m_1

(ii) output of node '3' should be '1', i.e., fault list 'K' should be effective.

(iii) but for fault list 'K' to be effective, output of node 5 should not change. In other words, fault list 'L' should be ineffective.

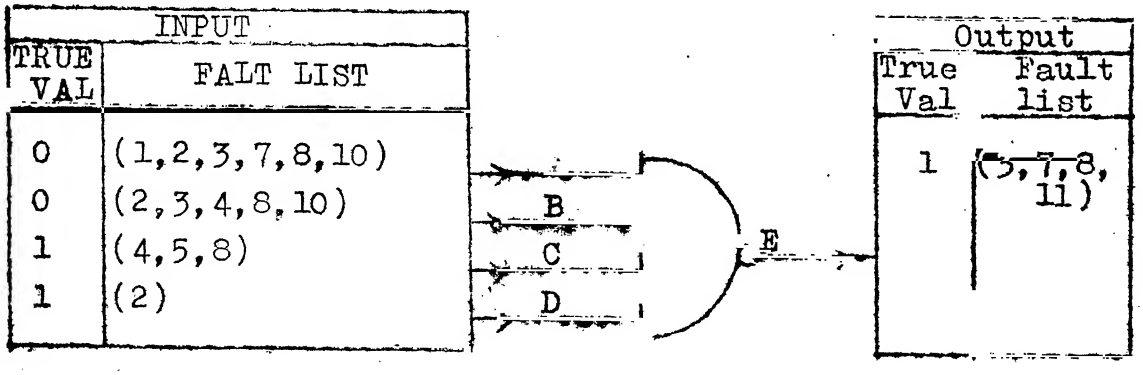
Conditions (ii) and (iii) above at the input of node 6 can be combined into a single condition, i.e., $(K \wedge \bar{L})$, and added to it the output fault of the node result in the fault list 'M'.

(e) An Approach to Algorithm Generation for Fault List Evluation (Refer

The example below illustrates the procedure for development of an algorithm for evaluation of fault-list for a NAND gate. Similar procedures can be made for other types of gates also.

Example

In the figure shown below, the internal faults of gate E are numbered from 6 to 10, and those carried from the previous stage (i.e., external to the gate) are numbered 1 to 6. The input fault lists associated with various pins are A,B,C,D with various faults specified against each as shown. Verify the output fault list by a procedure which can be implemented by the computer.



Faults internal to 'E' are:-

6 Input A open
7 Input B open
8 Input C open
9 Input D open
10 Output s-a-1
11 Output s-a-0

Notice that the input fault list contains some fault names on gate E (i.e., faults 7,8,10 on input A; 8 and 10 on input B, and fault 8 on input C), which were previously calculated and propagated to gate E again due to the presence of circuit feedbacks. Three observations are necessary to handle the internal faults:

(i) For a '0' input, the corresponding input open fault behaves as though it were always in that input fault list since it forces that input to value '1'. Thus faults 6 and 7 will be treated as though they were in the fault lists on inputs A and B respectively.

(ii) For a '1' input, the corresponding input open fault behaves as though it were never in that input fault list since it cannot force that input to the value '0'. Thus faults 8 and 9 will be treated as though these were not in the fault lists on inputs C and D respectively.

(iii) If the gate output value is 1/0, it must contain gate s-a-0/s-a-1 fault and cannot contain the gate s-a-1/s-a-0 fault. Thus fault No. 10 is removed from the output fault list and fault No. 11 is added to the output fault list.

With the incorporation of these observations into an algorithm, the output fault list should be worked out as

$$E = \{A \cup 6\} \cap \{B \cup 7\} \cap \{C \ominus 8\} \cap \{D \ominus 9\} \cup \{11\} \ominus \{10\} \quad \text{---(I)}$$

[Note: Notation '$\ominus$' represents difference of two sets]

$$= \{(1,2,3,6,7,8,10) \cap (2,3,4,7,8,10) \cap (1,2,3,6,7,8,9,10,11) \cap (1,3,4,5,6,7,8,9,10,11) \cup (11) \ominus (10)\}$$

$$= \{(2,3,7,8,10) \cap (1,3,6,7,8,9,10,11) \cup (11) \ominus (10)\}$$

$$= \{(3,7,8,10) \cup (11) \ominus (10)\} = \{3,7,8,11\}$$

In summary, the output fault list computation for a NAND gate with no unknown inputs and no star faults is described below:

(i) To account for the impact of internal faults, form 'temporary 0 lists' which are union of the '0 lists' and their corresponding input open faults. Form 'temporary 1 lists' which are the '1 lists' minus (i.e., SET DIFFERENCE) the corresponding input open faults.

(ii) Form the 'temp. output fault lists' from the 'Temp. 1 lists' and 'Temp. 0 lists' as follows:

1. If some inputs have the value 0 and the remainder have the value 1, the temp. output fault list is obtained by forming set intersection of the 'temp 0 lists' minus the set UNION of 'temp. 1 lists'. This is evident from Eqn.(I) since the same can also be written in the following form

$$E = \{(A \cup 6) \cap (B \cup 7)\} \ominus \{(C \ominus 8) \cup (D \ominus 9)\} \cup \{11\} \ominus \{10\}$$

2. If all the inputs have the value 0, the temp. output fault list is obtained by forming the intersection of all the 'temp 0 lists'.

3. If all inputs have the value 1, the temp. output fault list is obtained by forming the union of all the 'temp. 1 lists'.

4. Final output fault list from the temp. output fault list is obtained by merging (i.e., UNION) with it of the output fault that may change the true output value of the gate and by removing (i.e., DIFFERENCE) the other output fault from the temp. fault list.

(f) Why Call it a Deductive Technique?: Fault simulation approach can be either INDUCTIVE or DEDUCTIVE. In the INDUCTIVE method single permanent stuck-at faults are induced into the circuit and are propagated to the primary output point by various technique discussed earlier. In the latter approach, at every nodal point, after evaluating the true output value, a deduction is made as to which possible faults, upto that point, can alter the output value of the node. All such faults are grouped together to form a fault list. Since this deductive process is repeated till all nodes have been simulated and the circuit values have stabilized, this approach has been termed as the 'DEDUCTIVE' technique of fault simulation.

(g) Main Characteristics of the Deductive Simulator:

An implementation of the deductive simulator has following salient characteristics:

(i) It is table driven (not compiled) i.e., description of logic circuit to be simulated is stored in a tabular form in the host computer. These tables are accessed by the simulation program as necessary. Simulation program is circuit independent, but a new set of tables is required for each distinct circuit to be simulated. In contrast, it may be appropriate to mention here, the compiled simulators contain the description of the circuit implicitly in the simulation program thereby requiring recompiling for each distinct circuit.

(ii) It models the dynamic behaviour of the simulated circuit fairly closely i.e., it updates the computations at successive uniform intervals of simulated times 1T, 2T, 3T ... where T is equal to the average gate delay. This results in accurate time modelling to the extent that all gates have the same nominal delay T, and the variation in gate delays from the nominal value in the actual circuit is small, a condition which in-fact holds for many of the circuits being simulated. Through some simple changes, the simulator could be arranged to simulate a different delay for each gate, though at the expense of increased simulation time.

(iii) It employes the 'SELECTIVE TRACE' procedure. It leads to economy in simulation provided the logic nodes are simulated in the same order as they occur in the logic flow. The desired ordering is achieved by the use of a linked list structure for the tables describing the circuit. It ensures that the first set of nodes to be simulated are those connected directly to the primary input points, those to be simulated next are those fed by the first set, and so on. In this manner the propagation of signals through successive logic elements are simulated until all logic states have stabilized. If oscillating loops are present, in which case the circuit never stabilizes, the simulation can be terminated after an appropriate time specified in the simulator.

(iv) Three logic states are simulated, namely zero, one and DON'T KNOW. The latter state is useful when the complete internal state of the circuit is not known at the beginning of simulation, or the test conditions applied to the circuit are not fully specified. In brief, this 'don't know' state helps in initialization and facilitates simulation.

(h) Speed of Implementation: We have observed earlier in Section 'd' above that fault list evaluation primarily concerns with the implementation of basic set operations like 'set intersections', set unions', and 'set differences'. Predominant percentage of simulation time is occupied in performing these computations. The underlying operation common to all these set operations is the matching of fault enteries in two or more lists. A straight forward procedure for performing the matching operation on lists requires a number of computational steps proportional to the product of list lengths. The use of such a slow procedure would probably negate the potential speed advantage of the deductive simulation method. It is of utmost importance to use more efficient procedures for these set operations to reduce simulation time.

(i) Simulation of flip flops and other Complex Devices It is difficult compared to a gate to determine fault lists at the output of a flip flop since the latter can 'remember' the effects of faults which were propagated to its inputs at a previous time in a test sequence.

This we shall be discussing in detail in the last chapter.

4. COMPARISON OF PARALLEL AND DEDUCTIVE TECHNIQUES

Before we endeavour to compare these two simulation techniques, let us understand briefly the methodology of the PARALLEL SIMULATION approach.

Parallel fault simulation is the simulation in parallel of a number of copies of the object network. These copies are normally referred to as machines. One machine represents the fault-free network and is known as the good machine, and all other machines represent the object network with atleast one fault present. The number of machines that are simulated in parallel is normally constrained by the number of bits in the host computer word. Procedure followed may be as under:

(A) Initially all the faults to be simulated are stored in a 'Master Fault File' which may contain the following type of information:

(a) fault number

(b) fault type (i.e., 1 = s-a-1, 2 = s-a-0, etc.)

(c) signal position (i.e., -1 = signal fault, 0 = complex fault etc., indicating also the faulty pin number)

(d) Name of element

(e) Name of signal.

(B) During actual simulation, the records describing some number of faults to be simulated in parallel can be read from the master fault file, temporary fault tables can be created, and a simulation pass can take place.

(C) At the end of a fault simulation pass, additional number of faults can be read in and the same process contained until all faults are simulated.

(D) Now is the requirement of determining an algorithm for propagating a number of faults in parallel. This, in other words, is the selection of the data-structure which will be used to represent signal values. One possible data structure for 3-value signals (0,1,Don't know) in a parallel machine may be as follows:

(i) represent adjacent bit pairs for the value of a signal for one machine.

(ii) reserve the right most or left most bit pair for the good machine, and use other bit pairs for faulty machines.

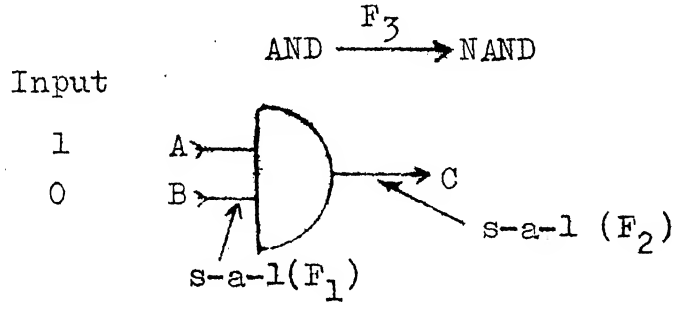
(iii) Let bit pairs 11 mean 1
00 mean 0
01 mean X (i.e., Don't Know)
10 mean either not used or represents X

(iv) number of machines which can be simulated

$$= \frac{\text{No. of bits in host computer}}{2}$$

Example

Consider an AND gate with the three faults shown in the figure below resulting in faulty machines F_1, F_2, and F_3. Make a data structure for simulation of 4 machines in parallel (including one good machine 'G')



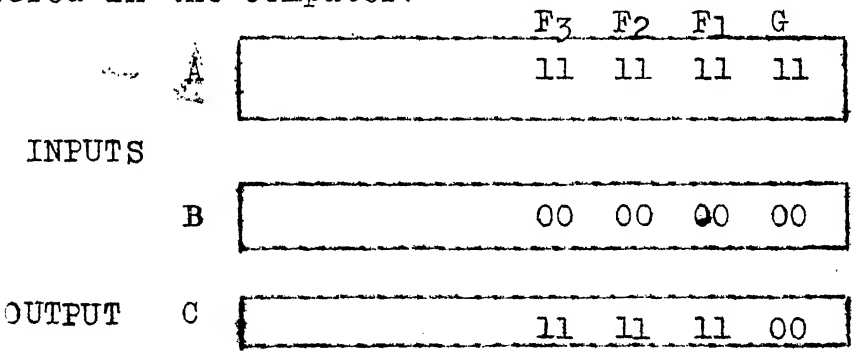
Type of faults are:

(i) Pin B s-a-1 faulty machine named F_1

(ii) Pin C s-a-1; faulty machine named F_2

(iii) Gate changes its function from AND to NAND; faulty machine named F_3.

For the input combination A,B = (1,0), and with fault insertion, the four machines will give following results stored in the computer:



With this much background of parallel fault simulation we attempt to compare this technique with the deductive approach. Following points may be considered:

[a] since information regarding all the faults in a circuit is first stored in a 'master fault' file' for parallel simulation, everytime the circuit description is modified this master fault file would have to be created anew. Creation of such a file is not necessary for deductive fault simulation. The type of faults to be simulated for different types of gates is pre-specified and the simulator automatically takes into account all these faults simultaneously for each of the nodes being simulated.

[b] the number of machines that are simulated in parallel is normally constrained in some way by the number of bits in the host computer word. A maximum of N faults can be simulated per run where N is the number of bits in the computer word. If a total of M faults are to be simulated, then M/N runs are needed to determine all detectable faults per input. In the case of deductive simulator only one simulation run is necessary for each applied test vector.

[c] the simulation time per run for the deductive simulator is probably much longer than the time per run for parallel simulator, but is expected to be much less than

for M/N runs which the parallel method requires to accomplish equivalent results. One reason for this potential time reduction is that deductive simulator performs computations on each fault along only the particular paths that are sensitized to propagate its effects. In contrast, the parallel method performs computations along all paths for which the effects of any one (or more) of the N faults in the batch being simulated continue to propagate. The latter may, therefore, perform much unnecessary computation.

[d] since the number of runs and hence the time for simulation increases with the increase in number of faults injected, the parallel simulator will probably be more effective for small size circuits with less number of associated faults. The deductive simulator, in comparison, will be far more effective for large size circuits with large number of faults to be simulated.

[e] a potential disadvantage of the deductive simulator is that it requires considerably more memory space in the host computer than does the parallel method. Also, the amount of memory needed is not accurately predictable prior to a run, so dynamic memory allocation becomes desirable. And because of this dynamic memory allocation using linked-list structures, the implementation of the deductive technique is a more complex process compared to parallel fault simulation.

[f] implementation of deductive technique needs development of special algorithms for fast set-operations. The speed of fault simulation is directly affected by the speed at which these operations can be performed. However, no such problem is encountered in parallel fault simulation.

CHAPTER 3

M P L E M E N T A T I O N O F D E D U C T I V E M E T H O D

CHAPTER 3

IMPLEMENTATION OF DEDUCTIVE METHOD

In this chapter we shall develop the SIMULATOR step by step indicating the approach actually followed. Flow chart analysis of the problem has been considered necessary to indicate the basic structure of the simulator 'program' so as to facilitate the reader in grasping easily the subsequent discussion.

1. ASSUMPTIONS AND APPROXIMATIONS

(a) It is assumed that interval of simulation is equal to the average gate delay, and variation in delays of different gates is too small to affect the results of simulation. Thus a 'unit-time delay' simulation model has been used.

(b) A 'Three-value' simulation model has been attempted. Besides '0' and '1' as the known binary values, an unknown value, represented by '2', has been used.

(c) Only 'single stuck-at faults' have been modelled. Some reasons for this assumption are as follows:

(i) frequent testing of a circuit usually implies the existence of not more than a single fault.

(ii) intermittent failures are, no doubt, prevalent in many technologies (i.e., TTL, DTL, MOS, etc.) but these are ignored with the assumption that such intermittent faults shall persist long enough so as to appear as a permanent fault and get detected ultimately.

Type of faults modelled are:

(i) Inputs of gates stuck-at non-dominant values (i.e., '1' for NAND/AND gates and '0' for NOR/OR gates. This simply amounts to the 'input terminal OPEN' condition for a gate.)

(ii) outputs of gates stuck-at-'1'/stuck-at-'0'.

(d) Only gate level simulation has been attempted. A maximum of 500 gates in a circuit have been catered for. Various gates simulated and the codes used for the same are given below:

(i)	NAND	= 2
(ii)	AND	= 3
(iii)	NOR	= 4
(iv)	OR	= 5
(v)	X-OR	= 6

(e) Maximum FAININ and FANOUT restriction on any gate has been kept as '8' and '10' respectively.

(f) Number of primary outputs of a circuit has been restricted to 50, though no limit has been set on the number of primary inputs.

(g) It is assumed that the circuit is irredundant since all stuck-at faults in a circuit cannot be detected if it contains a logical redundancy.

(h) A maximum time limit has been pre-set in the program for suspending further servicing of that test vector which generates oscillatory conditions in the simulated circuit model.

(j) No race analysis have been performed. Spikes and logical hazards in the circuit have not been modelled since these need a 'multiple value and variable time-delay' simulation model.

(k) Output pins of all gates have been numbered as '1' and the input pins start with serial No. '2' onwards and go upto serial No. '9' depending on the specified number of input pins of the gate actually used in the circuit. This numbering sequence has been followed for ease of programming.

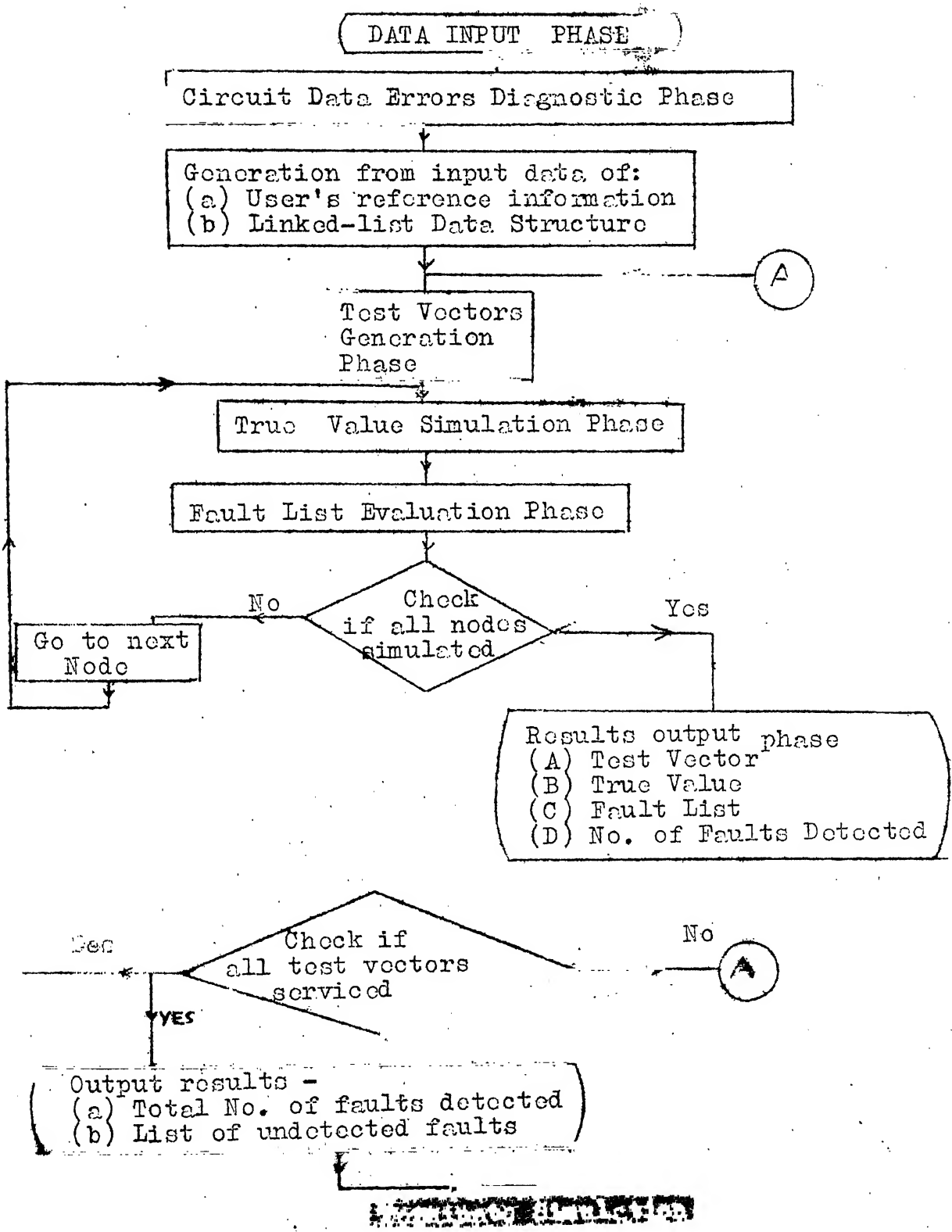
(l) Maximum number of faults in the fault-list of any node does not exceed 20.

Above mentioned approximations are to make a reasonable compromise between modelling accuracy and the modelling speed. Multi-valued and assignable time-delay models will surely add to the accuracy of simulated circuit's response, but not without increasing the complexity of the model resulting in extra simulation effort and higher time consumption.

2. PROBLEM ANALYSIS BY FLOW CHART METHOD

Our approach to this analysis will be to introduce first the 'BLOCK SCHEMATIC' of the simulation plan, and then to follow it up with a description of each individual block.

BLOCK SCHEMATIC



DESCRIPTION OF VARIOUS BLOCKS

(a) Data Input Phase

This phase does not need an illustration with a flow chart. However, following points need emphasis:

(i) a 'coding scheme' for different types of gates, as discussed previously, has been used. Code Nos. 7 to 10 have been reserved for future development for ELEMENT LEVEL simulation. '-1' is the code used for PRIMARY INPUTS to the circuit.

(ii) circuit data input is in a tabular form. The circuit to be simulated is converted into a tabular data structure by the method discussed in the next section. This data is then punched on the cards by the user as per the formats specified in 'Instructions to the User' section, and is stored in the main memory of the host computer.

(iii) input data consists of following information:

1. Gate Number
2. Code for the gate
3. No. of inputs of the gate actually used
4. Gate output loading figure
5. Gate output connections
6. Total number of gates in the circuit.

(iv) besides the circuit data, following information is also supplied by the user:

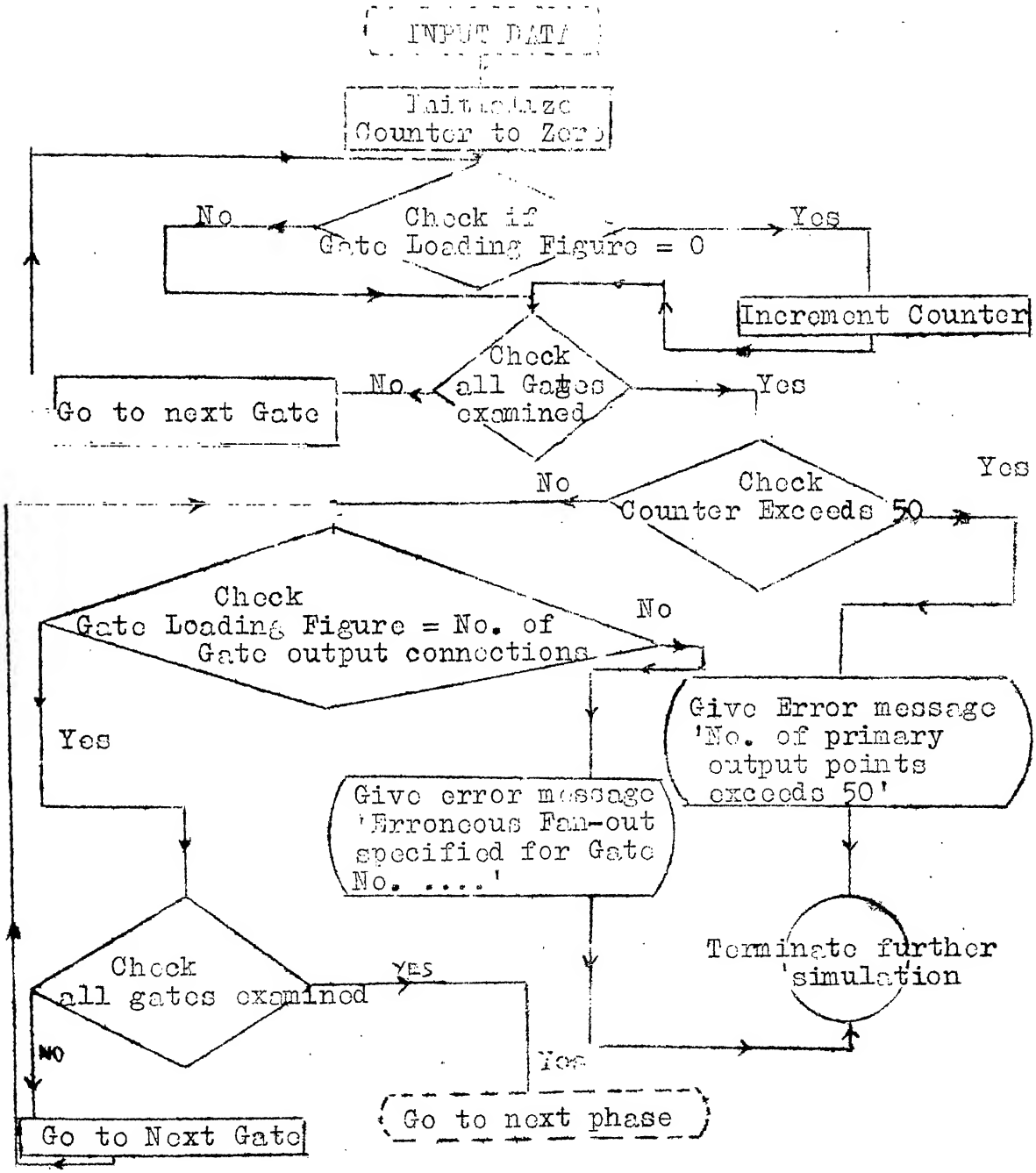
1. Type of simulation required (viz., only true value simulation or fault simulation)

2. Test points in the circuit where output is required to be monitored.

(b) Circuit Data Errors Diagnostic Phase

Before the logic simulation begins, it is necessary to verify correctness of the circuit data since in the process of tabulation some omissions are likely, especially for larger circuits. Such omissions cannot be easily

spotted by visual inspection of the table and the co-related circuit, and are thus included in this phase of computation. The procedure followed is given in the flow-chart below:



Besides the above error-messages, following types of errors are also diagnosed in the data in the later phases as the simulation proceeds:

(i) 'Fain-in of the gate exceeds maximum limit of 8'.

(ii) 'Undefined code used for a gate'. This error-signal is received if code value exceeds 10.

(iii) 'Reserved Code (Non-assigned)'. This error signal is received if code used is between 7 and 10 for any gate, and indicates that subroutines for true value and fault list evaluation for these codes are yet to be developed;

(iv) 'Oscillations encountered in circuit simulation'. Proceed to next test vector.

It may be observed that error message (iv) above is not directly related to circuit input data. In fact, it pertains to an error encountered in execution of the program.

(c) Generation of User's Reference Information and Linked List Data Structure

This information is generated to check and verify the correctness of those parameters which are later used for true value and fault simulation. Flow-chart discussion of this phase is not considered necessary, though the type of information generated is indicated below:

(i) Total number of primary inputs.

(ii) Total number of logic gates in the circuits.

(iii) Total number of expected faults in the circuit.

(iv) Primary output points.

(v) A linked-list structure indicating the connections of all input pins of various gates. We have named this list as 'INFORM' and it contains, in a serially ascending order, information about the gate numbers to which various input pins of any node have been connected. The LINK to this list has been called as 'PINLST' which locates the position of the first input pin of any gate in the 'INFORM' list.

An example may illustrate the point better.

Example

Consider the information generated below.

Gate No.	1	2	3	4	5	
PINLST	0	0	0	1	4	
INFORM	1	2	3	4	3	2

(i) Gates for which PINLST value = 0 are primary input points. In this example, such gates are Nos. 1,2 and 3.

(ii) Pin 2 of Gate 4 is connected to output of Gate 1. Similarly, Pin 2 of Gate 5 is connected to output of Gate 4. Logic followed is -

Gate 'B' to which input Pin 2 of Gate 'A' is connected = INFORM (PINLST (GATE 'A'))

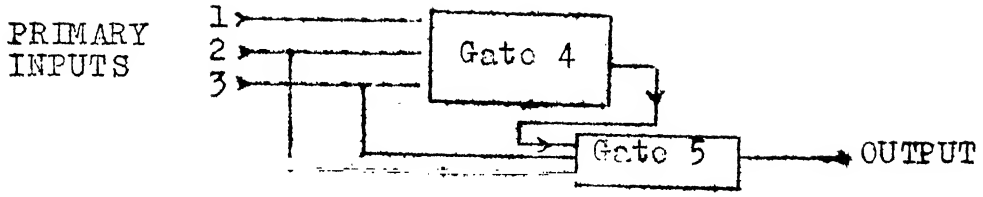
(iii) Balance of enteries in the INFORM list pertain to remaining pins of gates 4 and 5, viz.,

Pin Nos, 3 of Gates 4 and 5 are connected to Primary inputs 2 and 3 respectively

and

Pin Nos. 4 of Gates 4 and 5 are connected respectively to Primary inputs 3 and 2.

Thus the above linked-list structure represents the following circuit configuration.



The type of logic gate used is indicated by the CODE of the gate as referred to in the 'Circuit Data Input Phase' already discussed.

(vi) In a similar manner as above, another list called 'Record' stores, in a serial order, the status of all possible faults in a logic circuit. A gate-level link to this list is provided by the 'POINT' array which specifies the position in the 'Record' list of the first gate fault i.e., 'Pin 1, s-a-0'. This information, however, is not printed out.

Example

Let us consider the following information generated which is relevant to the circuit of the previous example:

Gate No.	1	2	3	4	5
Point	0	0	0	1	6
Record	1 0 0 0 0 0 1 0 1 0				

This implies the following:

1. Gate Nos. 1,2, and 3 for which POINT value = 0 have no fault (being primary input points).

2. Faults concerned with Gate 4 start from entry No. 1 of RECORD file, and those of Gate 5 start from Entry No. 6 in the RECORD file. (No. of gate faults for any gate cannot exceed '10' since a maximum of '8' faults can be at the input pins and 2 faults at the output pin.)

3. If the flag = 1 for any RECORD entry, it implies that this fault has been detected at the Primary output point.

4. Sequence of listing in the RECORD file for any gate starts with the output pin s-a-0 and may be summed up by the algorithm -

   Entry No. of output pin of gate 'G' s-a-0 in RECORD file = POINT (G)

   * RECORD (1) means output Pin of Gate 4, s-a-0
   ** and since its value = 1, this fault has been detected at the output.

Similarly, other faults pertaining to Gate 4 are

RECORD(2) - Output Pin s-a-1
RECORD(3) - Pin 2 s-a-1/s-a-0
RECORD(4) - Pin 3 s-a-1/s-a-0
RECORD(5) - Pin 4 s-a-1/s-a-0

Type of fault depends on type of gate used. These all will be s-a-1 type if Gate 4 is NAND/AND gate.

5. We may likewise interpret that 'output pin of gate 5, s-a-0' fault starts from Entry No. 6 of the RECORD file.

Verify that the remaining faults detected at the output are:

'Output pin of gate 5 s-a-1'
and 'Pin 3 of Gate 5 s-a-1/s-a-0'

(This is left as an exercise to the reader!)

(d) Test Vector Generation Phase

We have either generated the test vectors randomly or given it as an option to the user to specify his own test vectors. This will be discussed in Section 7 of this chapter.

Many different strategies could be used for generating random test sequences. We have restricted ourselves to the K-th input pattern to differ from (K+1)-th in just one bit position. Maximum number of test vectors to be generated = $(2)^N$ where N is the No. of primary inputs to the circuit. Because of some non-linearity in the functioning of the computer in-built random number generator, some test-vectors get repeated. This non-linearity is due to the probability density distribution being non-uniform over a period of time. Since about 60 percent of the faults get detected during the first 30 to 40 percent of the test

vectors serviced, when there is negligible repetition of test vectors, we have not attempted to study the problem deeper. Moreover, we have provisioned for termination of further simulation in case no new faults are detected by the servicing of ten successively generated test vectors. This exercises considerable economy on simulation time.

(e) True Value Simulation Phase (See flowchart on next page)

(i) To start with we initialize the output of all nodes to the unknown, value, '2'.

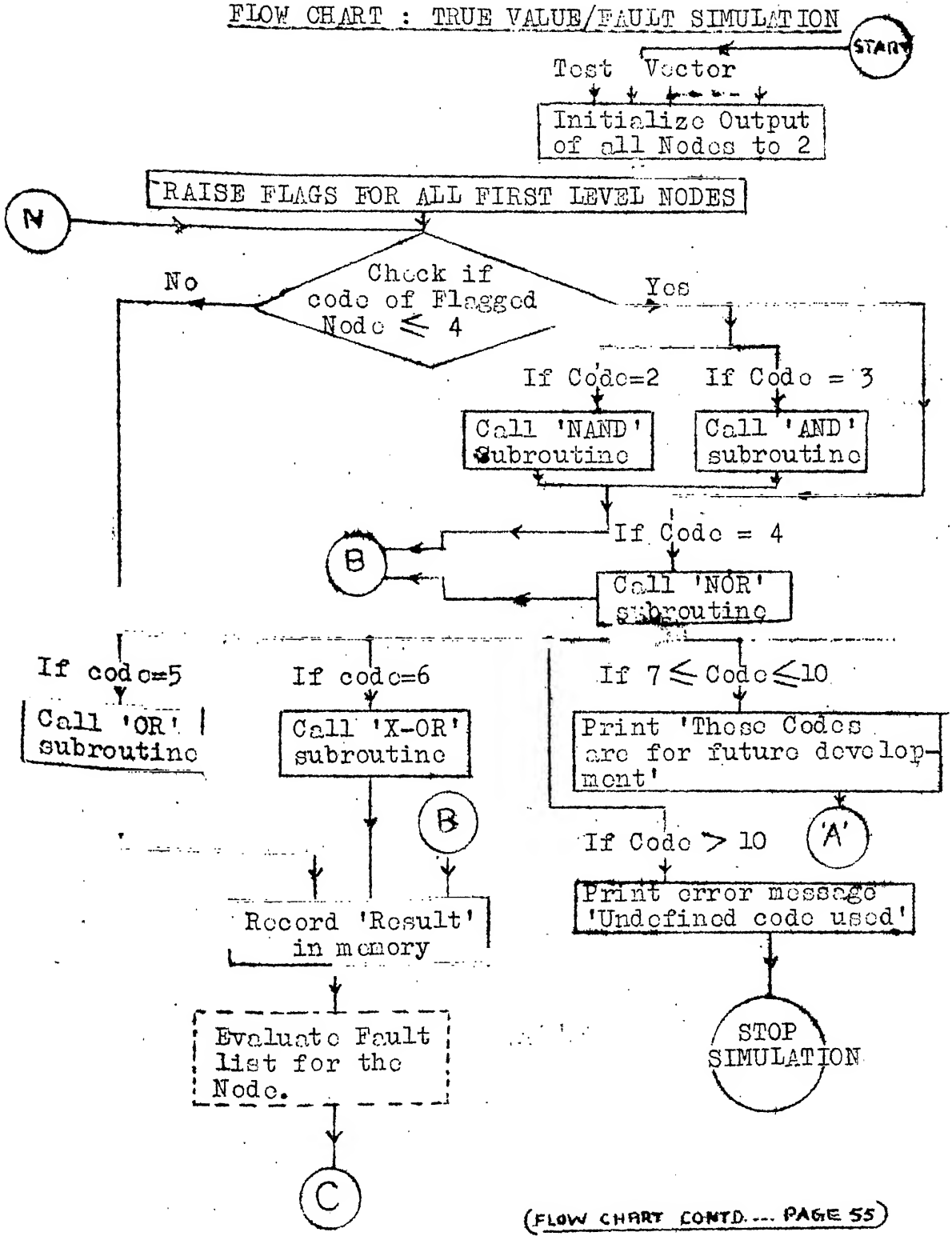
(ii) Raising of a 'flag' for a node means that that node is to be evaluated.

(iii) True value determination and fault list evaluation for any node are done concurrently. So flow-chart for 'Fault Simulation Phase' shall be common to that for the 'true value Simulation Phase'. However, if the user is interested in only TRUE VALUES at the output points, for purposes of logic verification of his circuit, he can exercise this option in the program. This will be discussed in the last section of this chapter.

(f) Fault List Evaluation Phase

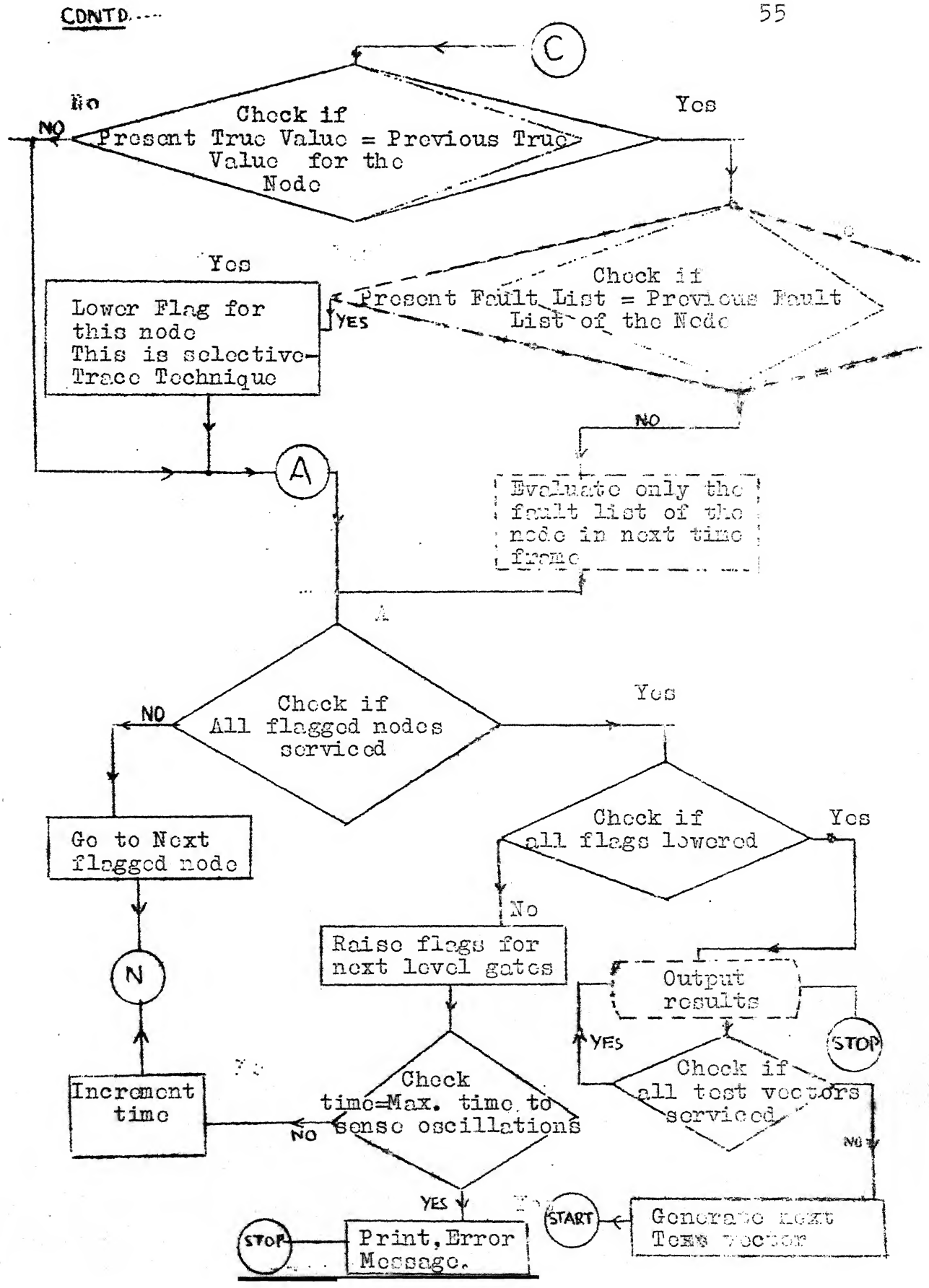
As the program, during execution, enters this phase of simulation it determines the set of faults which propagate to the output of a particular node for the given test vector. Some algorithms for fault-list evaluation have been discussed later in this chapter

FLOW CHART : TRUE VALUE/FAULT SIMULATION



(FLOW CHART CONTD.... PAGE 55)

CONTD....


C
No
NO
Check if Present True Value = Previous True Value for the Node
Yes
Yes
Lower Flag for this node This is selective-Trace Technique
YES
Check if Present Fault List = Previous Fault List of the Node
NO
Evaluate only the fault list of the node in next time frame
A
NO
Check if All flagged nodes serviced
Yes
Go to Next flagged node
Check if all flags lowered
Yes
No
Raise flags for next level gates
N
Output results
YES
STOP
Check if all test vectors serviced
Increment time
Check time=Max. time to sense oscillations
NO
NO
YES
START
Generate next Test vector
STOP
Print, Error Message.

under 'FAULT SIMULATION PROCEDURE'. The fault lists for different nodes are stored in time-set arrays (i.e., dynamic allocation of computer memory space) called 'FLIST'. Only two successive time-frames are recorded in these lists and a link to these lists is provided by another dynamic record called 'LIST'. This is similar to the linked-list structure discussed previously in Section C.

Fault lists of a node for two successive time intervals are compared to assess their stabilization. If the fault lists happen to be the same, it is apparant that the true output values of the node in the same time interval will always be the same. Flag for this node is thus lowered meaning thereby that the 'fan out' of this gate need not be evaluated in the next time frame. This is the 'selective trace technique' implementation.

3. TABULAR REPRESENTATION OF THE LOGIC CIRCUIT

In this section we shall discuss how a given logic circuit is to be converted into a tabular form for feeding the circuit data to the computer. Various steps involved are as follows:

(a) Starting with the primary inputs, serially number all the different nodes. This numbering does not take into account any logic-levelling of the circuit. It can be arbitrary and in any order.

(b) Output pin of each node will be called Pin No. 1, and the other pins, starting from top-most pin of the gate, will bear pin Nos. 2 to 9 for a maximum Fainin = 8 for any gate. These pin numbers need not be physically marked. Their numbering will be considered as an implied sequence followed throughout the program. Recognition of any pin in the circuit will be provided by the use of real numbers. For example number 6.3 will indicate pin 3 of node No. 6.

(c) Now we start with node No. 1 onwards and repeat following steps till data is tabulated for all the nodes:

(i) Check CODE allotted to the node from the coding scheme used. Enter it under the column 'CODE'.

(ii) Count number of pins of the node actually used in the circuit. Record these under the column 'INPUTS'. For Primary inputs this value is '0'.

(iii) Check actual loading of the node (i.e., number of places to which output of a node is physically connected). Record this in the column 'FANOUT'. For Primary output nodes, this value = 0.

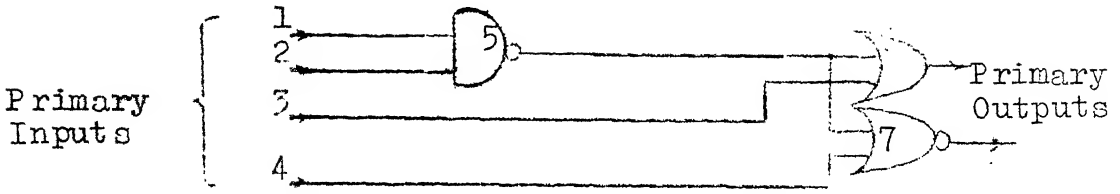
(iv) Check the FANOUT details (i.e., the pin number and gate number of the load) and record these under the columns of 'GLINK' (which means - GATE TO WHICH LINKED). Enter '1000.' as the last entry in the GLINK for each node. This is for ease of programming. In case of primary output points, only entry under GLINK will be 1000.

(v) This completes the data for a node.

We shall illustrate the above scheme with the help of the following example:

Example 1

(a) Circuit Diagram



Gate numbering has been done as shown in the Figure.

(b) Tabular Representation of the Circuit

Gate No.	Code	Inputs	Fanout	GLINK
1	-1	0	1	5.2,1000.
2	-1	0	1	5.3,1000.
3	-1	0	1	6.3,1000.
4	-1	0	1	7.3,1000.
5	2	2	2	6.2,7.2,1000.
6	5	2	0	1000.
7	4	2	0	1000.

It may be observed that this conversion of logic circuit into the above data structure is fairly easy and needs no technical skill irrespective of the length of the circuit involved. This type of tabular representation was developed to do away with the 'PRE-PROCESSOR' requirements for generation of data tables as discussed in some of the papers published on the subject (Refer

To get a better understanding of this type of data representation, the user is suggested to draw the actual circuit configuration from the table given below:

Example 2

Gate No.	Code	Inputs	Fanout	GLINK
1	-1	0	1	5.2,1000.
2	-1	0	2	5.3,6.3,1000.
3	-1	0	2	5.4,8.3,1000.
4	-1	0	1	7.4,1000.
5	2	3	2	6.2,7.2,1000.
6	4	2	1	7.3,1000.
7	3	3	1	8.2,1000.
8	6	2	0	1000.

(Compare with the circuit diagram on page 81 and verify!)

4. DEVELOPMENT OF TRUE VALUE SIMULATOR

We have already described in Section 2 the broad outline of true value simulation technique by Flow chart analysis. Now we shall illustrate the development of a program for the same.

Besides the user's supplied data tables, we shall be using 'PINLST' and 'INFORM' lists generated already by the program. Since we decide to be concerned with only two successive time frames for our computations (viz., the 'current time' and the 'future time'), we shall be using two different time-set arrays called 'FLAG' and 'SETVAL'. These arrays shall store respectively the 'node evaluation status' (i.e., FLAG value = 1 meaning that node is to be evaluated), and 'node evaluation

results' (i.e., true value at the output of a node). The raised flags (i.e., FLAG = 1) in the 'current time' record act as a pointer to those nodes which are to be evaluated at the current time.

(a) Algorithms Used - To start with, flags are raised for all primary inputs (i.e., nodes with CODE = -1). Then either the user-supplied 'test vector' is read in and recorded in the 'SETVAL' current time array, or the test vector is randomly generated and recorded likewise. We are now ready to proceed with true-value simulation for the test vector that has been specified. But which nodes are to be evaluated (i.e., the first level nodes) must be entered in the 'future time' record. For this purpose FLAG array for the future time is used. Information about the nodes to be flagged is generated as per the Algorithm 'A'.

(i) Algorithm 'A'

(a_1) sense the current time flags starting with Node No. 1. If FLAG = 1, read various columns of 'GLINK' and proceed till 1000. is struk.

(a_2) every entry in GLINK column will be a real number with its mantissa part specifying the node no. for which flag is to be raised. Separate the mantissa part and raise the future time evaluation flag accordingly. Also transfer the 'SETVAL' record from current to the future time array.

(a_3)if GLINK = 1000., go to next node and repeat the above steps till all nodes have been examined.

Now our 'future time' record becomes the'current time 'record. We may thus proceed further with simulation time (called 'TIME') initialized to zero. This 'TIME' will be incremented with every single simulation pass through the circuit. It is likely that, during simulation, the circuit model enters into oscillations. An exit is to be provided for such a condition and is done in the program by providing a maximum limit of time (called MAXTIM) for which simulation may continue. If this time gets exceeded, the program jumps to the next test-vector. Let us assume that the oscillatory conditions do not arise. Then Algorithm 'B' evaluates the true value.

(ii) Algorithm 'B'

(b_1)start with node 1.

(b_2)if flag = 1 proceed further, else go to the next node and repeat from 'b_2' onwards.

(b_3)depending on the CODE of the node, enter in the 'FANIN' array non-dominant values for the node (viz., '1' for NAND/AND and '0' for NOR/OR). 'FANIN' array is used to record the actual pin values at the input pins of a node.

(b_4) check pin connections of the node from 'PINLST' arrays, and transfer from the SETVAL array all input pin values into FAININ array.

(b_5) go to the concerned subroutine for evaluation of the node output.

(b_6) enter the node evaluation results in the 'current time' SETVAL record.

(b_7) compare the two time arrays of SETVAL. If the true value is same (but not = 2), then put the current time flag for the node = 0 so that its fanout nodes are not evaluated next time. This is the 'SELECTIVE TRACE TECHNIQUE', otherwise keep the flag = 1.

(b_8) go to the next node and repeat from 'b_2' onwards till all gates have been evaluated.

(b_9) sense the 'current time' flags for all the nodes. If all the flags are lowered (i.e., = 0), print out results and go to the next test vector, else reset the future time flags to '0', and repeat from Algorithm 'A' above.

We have so far discussed the algorithms for true value simulation. Let us be clear that the program after step 'b_6' of algorithm 'B' shall normally proceed to the 'Fault Simulation Phase' and then return to step 'b_7', unless otherwise desired by the user. We mean that the user has to specify through a data card (discussed later in Section 7) if he needs the 'true value simulation only'.

Let us now understand the implementation of the above algorithms with the help of an example.

Example

Consider the circuit on Page 58 and its tabular representation.

For this circuit following information will be generated by the computer:

No. of Primary Inputs = 4
Primary Output Points are Gate Nos. 6,7

Gate No. -	1	2	3	4	5	6	7
PINLST -	0	0	0	0	1	3	5
INFORM -	1	2	5	3	5	4	

Let us work out the true value at various nodes for the user supplied test vector '1111'. This we shall evaluate for each time frame till the output at all the nodes stabilizes. That exactly shall be the manner in which computer processing takes place.

Time Frames	Gate No.	Flag		SETVAL	
		Current Time	Future Time	Current Time	Future Time
TIME = 0	1	1	0	1	1
	2	1	0	1	1
	3	1	0	1	1
	4	1	0	1	1
	5	0	1	2	2
	6	0	1	2	2
	7	0	1	2	2
TIME = 1	1	0	0	1	1
	2	0	0	1	1
	3	0	0	1	1
	4	0	0	1	1
	5	1	0	2	0
	6	1	1	2	1
	7	1	1	2	0
TIME = 2	1	0	0	1	1
	2	0	0	1	1
	3	0	0	1	1
	4	0	0	1	1
	5	0	0	0	0
	6	1/0	0	1	1
	7	1/0	0	0	0

Note: FLAG VALUE 1/0 against Gate Nos. 6 and 7 indicate that the flags have been lowered due to selective trace procedure.

We observe that all the flags get lowered by the end of time Value = 2, and the nodes' output values stablize. Result at the Primary output points will be

Gate No.	True-Value
6	1
7	0

Note: Since this very circuit has also been simulated by the computer, these manually attempted results may be compared and verified from the computer output attached as Appendix 'A'.

(b) Subroutines Structure: For various types of gates simulated by the program, we have structured the corresponding 'gate evaluation subroutines'. The basic algorithms for these subroutines have been derived from the '3-value truth tables' for these gates which are given below assuming 'two-input' gates only. These algorithms have then been generalized to cater for maximum FAININ = 8 for any gate:

INPUTS		OUTPUT OF GATES				
A	B	NAND	AND	NOR	OR	X-OR
0	0	1	0	1	0	0
0	1	1	0	0	1	1
0	2	1	0	2	2	2
1	1	0	1	0	1	0
1	2	2	2	0	1	2
2	2	2	2	2	2	2

The 'INVERT' subroutine works on the logic that invert of 2 = 2 itself, while for '0' its value = '1' and vice-versa.

5. FAULT SIMULATION PROCEDURE

Since the fault-simulation has to be essentially preceded by true value simulation, the procedure discussed in Section 4 above is followed upto step 'b_6' of Algorithm 'B', and then it deviates to determine the FAULT LIST At that node. Some essential points are enumerated below before discussing the algorithms actually used.

(a) Essential Points

(i) Fault representation as well as processing in the computer has been done with the use of REAL NUMBERS. For example, a number 6.31 shall indicate that 'pin 3 of node 6 is s-a-1'. Similarly, a number 9.10 implies 'pin 1 of node 9 s-a-0'.

(ii) At the input pins only the non-dominant stuck-at faults and at the output pins both 's-a-1' and 's-a-0' faults have been modelled. Non-dominant faults are those which, when present at the input pins, do not stop the propagation of fault lists through the node.

(iii) Fault lists at various nodes are dynamically stored in a time-set array called 'FLIST', a link to which is provided by another similar time-bound list called 'LIST' which gives location in 'FLIST' of the first fault of the fault list pertaining to a particular node.

(iv) Status of various faults detected is maintained (as discussed earlier) in a list called 'RECORD', a link to which is provided by another list called 'POINT'. Generation of these lists was necessary to ascertain the number of new faults detected per test-vector. Also these lists provide information about the various faults which remain undetected after all the test vectors have been serviced.

(v) In true-value simulation we had used FLAG value either '1' or '0' indicating respectively whether a node is to be simulated or not. Now we include another value for the flag (i.e., = 2). Reason for this is that although true value for a node may stabilize in 'n' time frames, the fault list for that node may require 'm' more time frames to stablize. In such a case we should only revaluate that node's fault list in the next time-frame and not the true value. This is indicated by putting the FLAG = 2. Thus our revised implication of flag values will be as under:

FLAG = 1 means both true value and the fault list for the node are to be determined.

FLAG = 2 means only fault list is to be computed.

FLAG = 0 means neither fault list nor true value is required.

(v) In case any input pin value for a node = 2 that pin is ignored while evaluating fault list at that node. Such a condition results in extra time frames needed for fault list stabilization discussed above. If the true value for a node = 2, fault list at that node is not computed at all, and the control is transferred to the next flagged node.

(b) Extension of Algorithm 'B' of Section 3: In case control is transferred to the 'fault simulator' from Step 'b_6' of Algorithm 'B', following steps should be added to the sequence at that point:

$b_{6.1}$ Evaluate fault list at the node as per the algorithms discussed later, and record it appropriately in 'FLIST'.

$b_{6.2}$ Compare the node fault lists at two successive time-intervals. If same, reset the flag to '0' and return to Step 'b_8'. This is the selective trace technique. If not, set FLAG=2 and continue.

$b_{6.3}$ Compare the true values for the node at two successive time-intervals. If these are the same, retain the flag status (=2) which shall transfer control in next time frame to only fault-simulation for this node. If not, put FLAG=1 and return to 'b_8'.

(c) Algorithms for Fault List Evaluation for Different Types of Gates

The technique for deriving an algorithm for a NOR gate has been explained previously in Chapter 2 (Section 3 - d). We have extended that concept and worked out parallel algorithms for all other types of gates as well. These algorithms have been arranged for different types of input combinations to a node in a manner as to facilitate their computer programming. The general notations followed are given in the 'Index ' below:

(i) Index to Notations Used

F : Fault list at the output of a node.

F_i : Fault list at the i-th input pin.

F_j : Fault list at the j-th input pin.

G_{10} : Node output s-a-0 fault.

G_{11} : Node output s-a-1 fault.

$\wedge^K$: Intersection of all fault lists at pins with true value = K

U^K : Union of all fault lists at pins with true value = K

K : Binary value 0 or 1.

$\ominus$: Set difference.

G_i : Node's internal fault at the i-th input pin.

G_j : Node's internal fault at the j-th input pin.

i,j : Variables used for input pins with values ranging between 2 and 8.

(ii) Algorithm for NAND/AND Gates Mixed Inputs

Mixed Inputs '0' and '1'

$$F = \left\{\bigcap^{0}(F_i \cup G_i)\right\} \ominus \left\{\cup^{1}(F_j \ominus G_j)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ For NAND gate.

$$= \left\{\bigcap^{0}(F_i \cup G_i)\right\} \ominus \left\{\cup^{1}(F_j \ominus G_j)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For AND gate.

Only '0' values at all input pins

$$F = \left\{\bigcap^{0}(F_i \cup G_i)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ For NAND gates.

$$= \left\{\bigcap^{0}(F_i \cup G_i)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For AND gates

Only '1' values at all input pins

$$F = \left\{\cup^{1}(F_j \ominus G_j)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For NAND gates

$$= \left\{\cup^{1}(F_j \ominus G_j)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ for AND gates.

(iii) Algorithms for NOR/OR gates

Mixed inputs '0' and '1'

$$F = \left\{\bigcap^{1}(F_j \cup G_j)\right\} \ominus \left\{\cup^{0}(F_i \ominus G_i)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For NOR gates.

$$= \left\{\bigcap^{1}(F_j \cup G_j)\right\} \ominus \left\{\cup^{0}(F_i \ominus G_i)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ For OR gates.

Only '0' value at all input pins

$$F = \left\{\cup^{0}(F_i \ominus G_i)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ For NOR gates.

$$= \left\{\cup^{0}(F_i \ominus G_i)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For OR gates

Only '1' value at all input pins

$$F = \left\{\bigcap^{1}(F_j \cup G_j)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$ For NOR gates

$$= \left\{\bigcap^{1}(F_j \cup G_j)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$ For OR gates.

(iv) Algorithms for X-OR Gates

Mixed Inputs '0' and '1'

$$F = \left\{\bigcap^{1}(F_j \cup G_j)\right\} \ominus \left\{U^{0}(F_i \cup G_i)\right\} \cup \{G_{10}\} \ominus \{G_{11}\}$$

Only '0' Value at Both Input Pins

$$F = \left\{U^{0}(F_i \cup G_i)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$

Only '1' Value at Both Input Pins

$$F = \left\{\bigcap^{1}(F_j \cup G_j)\right\} \cup \{G_{11}\} \ominus \{G_{10}\}$$

(d) Algorithms for Set Operations: We have seen above that the basic set operations needed are 'set union', 'set intersection' and 'set difference'. All these operations need $(n)^2$ number of comparisons (i.e., of each element of one set with all the elements of the other set), which is quite uneconomical in processing time especially when a large number of such operations are required every time a fault-list is to be evaluated. We have attempted to reduce the number of comparisons by ensuring that all the faults in a set are always arranged in an ascending order both at the time of their generation and during the set operations. Moreover, the last element of any set is followed by (0.0) for further reduction of simulation time. Since we have assumed that maximum number of faults in the fault list of any node is limited to 20, the dimension value of all temporary storage allocations has been accordingly restricted to 20.

Let us assume we are to perform set operations on two sets 'A' and 'B', and the results are to be stored in the set 'C'. Following algorithms explain the procedure adopted:

(i) Algorithm for SET UNION operation (i.e., AUB) (This means include in 'Set C' all non-repeated elements of both sets A and B)

1. Initialize i = j = 1
2. Compare i-th element of A and j-th element of B.
3. If unequal,checks if either of the elements is zero. If so, transfer all elements of the other set into Set C and RETURN, otherwise continue. If the two-elements are equal, go to Step 5.
4. Check if the i-th element of A is smaller than the j-th element of B. If so, store i-th element of A in C and increment 'i'. Go back to Step 2 above and continue.
5. Store either of the elements in Set C. Check element value. If = 0., stop and return, otherwise increment both i and j (i.e., go to next elements of sets A and B) and go back to Step 2 above.

(ii) Algorithm for SET INTERSECTION Operation (i.e., A ∩ B) (This means include in Set 'C' only those elements common to both sets A and B)

1. Initialize i = j = 1.
2. Compare i-th element of Set 'A' with j-th element of B.

3. If equal, store either of these in C. Check the element value. If = 0., stop and RETURN, otherwise continue. If the two elements are unequal, go to Step 5.
4. Increment both i and j and go back to Step 2 above.
5. Check if i-th element of A is smaller than j-th element of B. If so, increment i and go back to Step 2.

(iii) Algorithm for SET DIFFERENCE Operation (i.e., $A \ominus B$)

(This means eliminate from Set A all elements of Set B, and transfer the balance to Set C)

1. Initialize i = j = 1.
2. Compare i-th element of A with j-th element of B.
3. If equal, check element value. If = 0., Stop and RETURN, otherwise increment both i and j and go back to Step 2 above. If the two elements are not equal, continue.
4. Check if the i-th element of A is smaller than j-th element of B. If so, store i-th element of A in C. Check if this element is 0. If so, Stop and RETURN, otherwise increment i and go back to Step 2 above. If i-th element of A is not smaller than j-th element of B, continue.
5. Increment j and go back to Step 2.

[Note: In all the above algorithms increments value is unity]

6. RESULTS and DISCUSSIONS

We have carried out fault simulation on the two circuits discussed previously under Section 3, Examples 1 and 2 (Pages).

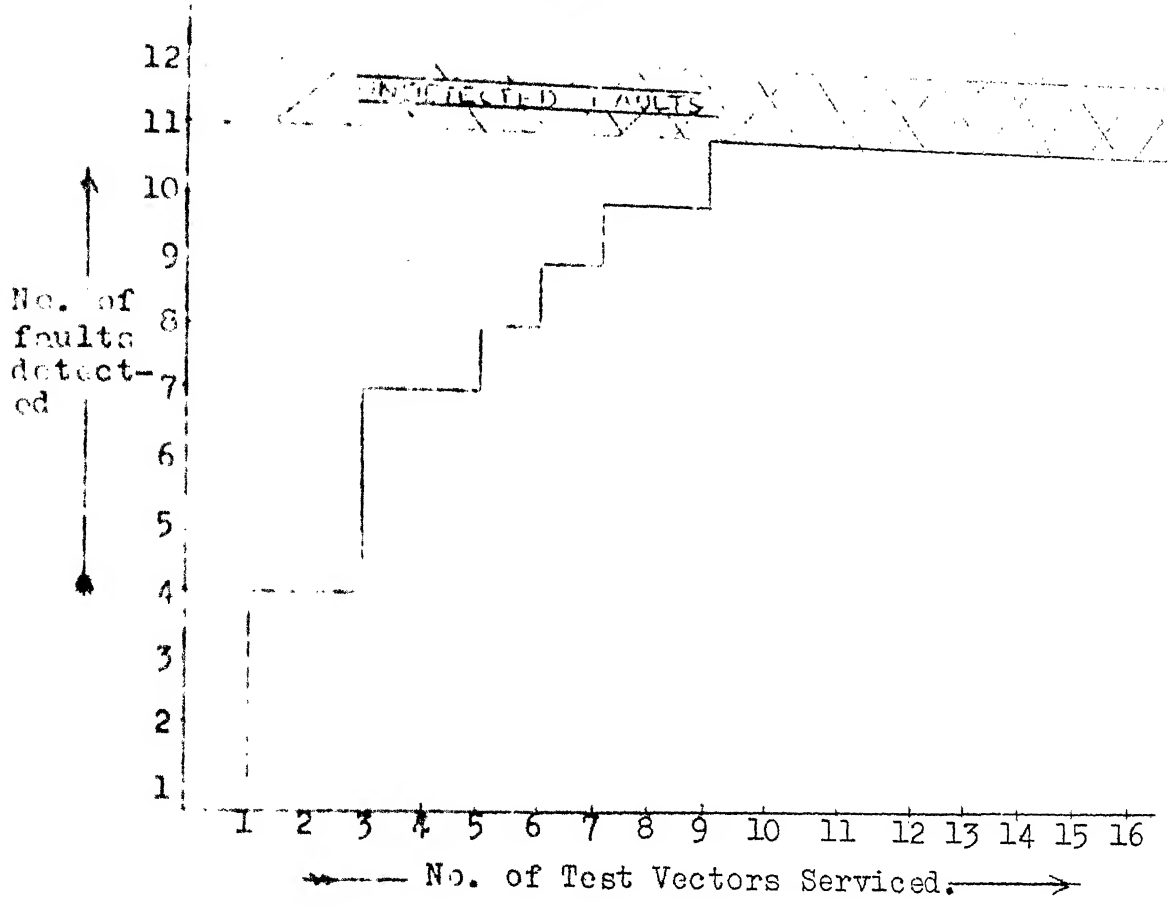
(a) Circuit of Example 1

This simulation has been done using the FORTRAN compiler. Results obtained are as under:

(i) No. of gates simulated = 3
(ii) No. of output points = 2
(iii) No. of faults in the circuit = 12
(iv) No. of test vectors required = 16
(v) No. of faults detected = 11
(vi) Percentage faults detected = 91.6 percent
(vii) No. of faults not detected = 1
(viii) Program compilation time = 4 Mts., 10 secs
(ix) Fault simulation time = 11 secs.

(x) Number of test-vectors serviced	No. of faults detected
1	4
3	7
5	8
6	9
7	10
9	11
16	11

Graphic study of Results

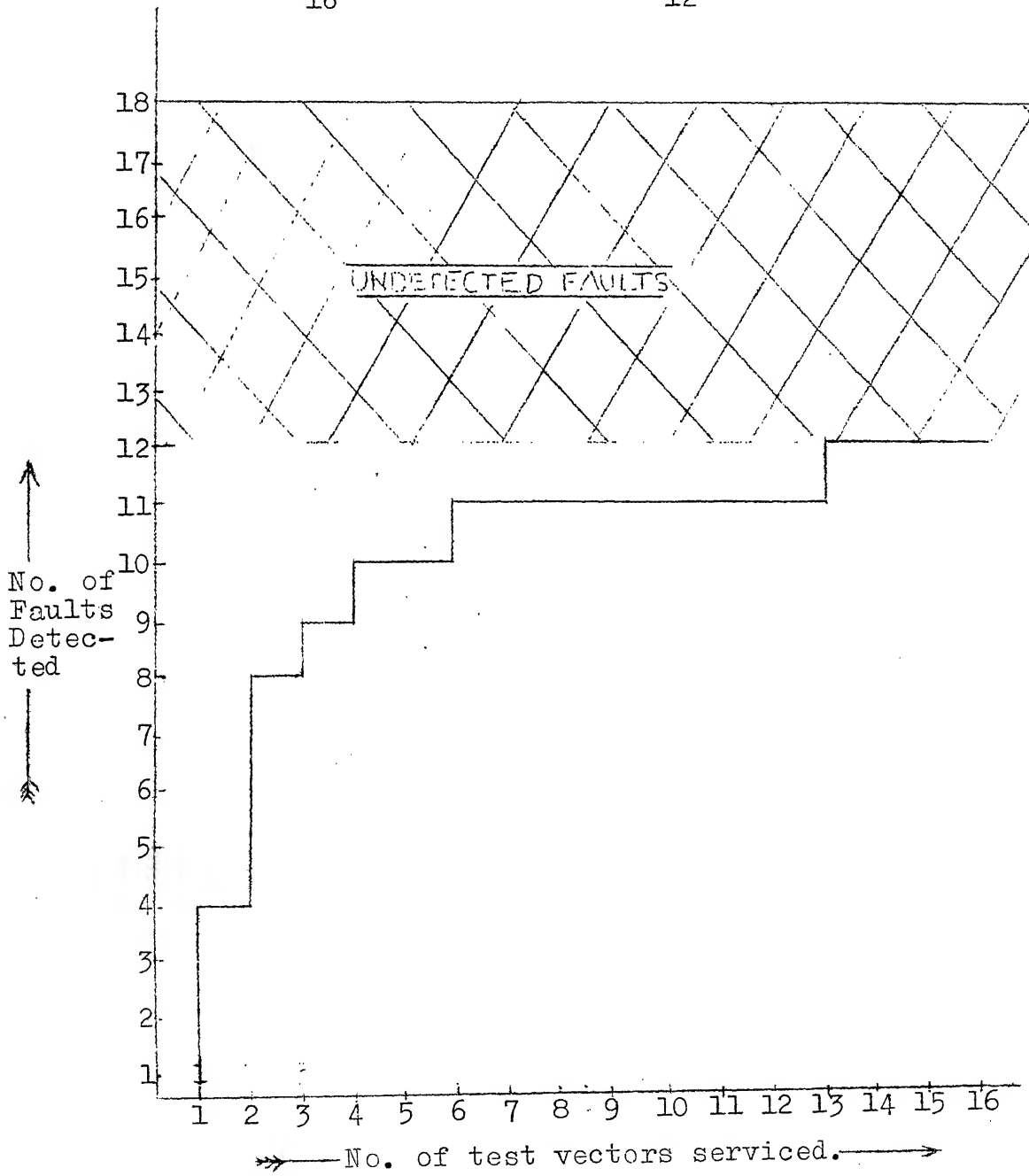


(b) Circuit of Example 2

This simulation has been done using the WATFOR compiler which is reportedly faster in compilation but slower in execution than the FORTRAN compiler.

(i) No. of gates simulated = 4
(ii) No. of output points = 3
(iii) No. of faults in the circuit= 18
(iv) No. of test vectors required= 16
(v) No. of faults detected = 12
(vi) Percentage of faults detected = 66.7 percent
(vii) No. of faults not detected = 6
(viii) Program compilation time = 15.4 secs.
(ix) Fault simulation time = 15.2 secs.

(x) No. of test vectors serviced	No. of faults detected
1	4
2	8
3	9
4	10
6	11
13	12
16	12



(c) Discussion and Conclusions from Above Results

(i) It is observed that more than 50 percent of faults get detected with the first 25 percent of the test vectors that are serviced.

(ii) The detection efficiency then suddenly reduces and another 20 to 40 percent of faults (depending on circuit configuration) get detected with the next 20 to 30 percent of the test-vectors serviced.

(iii) Thereafter the detection efficiency falls almost to zero, and the stage of saturation reaches when no further faults can be detected by this method.

(iv) It becomes advisable then to proceed with the 'D-Algorithm' or some other technique of fault simulation.

(v) Considering that the speed of execution for WATFOR compiler is double of that for FORTRAN compiler (which is a reasonable assumption based on statistical averages), it is observed that the simulation time for circuit of Example 2 should reduce to about 12 seconds if a FORTRAN compiler were used for the same. This period of 12 seconds includes the extra 5 seconds for higher dimension of variables in case of FORTRAN compiler. Our program has a dimension ratio of 10:1 for the data variables as programmed with these compilers. This restriction was to cater for the limited memory size of the WATFOR compiler. Since the number of gates as well

as the number of faults in this circuit are more than those for the circuit of Example 1, it can be deduced that speed of simulation increases with circuit size. This is because of 'selective trace technique'. In fact, this is one of the advantages of the deductive method which, as we said in Chapter 2, is faster for large size circuits. We have, however, no data available for comparison with the 'parallel technique of fault simulation'.

7. FACILITIES AND INSTRUCTIONS FOR THE USER

In this section we think it worthwhile to summarize the various facilities provided to the user of this simulator, and to list out some instructions on 'how to use these facilities?'

(a) Facilities for the User

(i) it is possible to use the simulator either purely for 'true value simulation' or for 'fault simulation' (which includes the true value simulation also)'. The former shall be useful for logic verification of a combinatorial circuit and the latter for purposes of fault diagnostics.

(ii) User can specify some of the test points in the circuit where he desires to have the output values. This number is, however, limited to '20' utmost.

(iii) Since the program registers only those nodes as primary output points which have FANOUT value = 0, the user has the liberty to specify additional primary output points subject to the condition that maximum number of output points does not exceed 50.

(iv) In case it is desired to have simulation results for particular test vector/vectors, the user can exercise this option by either specifying all the primary input values or by specifying only some of these.

(v) User can develop the 'element level' sub-routines, for the unused codes 7 to 10 and insert these appropriately in the program to increase the scope of utilization of this simulator.

(vi) The circuit data in tabular form and some information that gets generated from this data for user's reference are printed out by the program. This is for cross-check and verification of those circuit parameters which, though in error, are not spotted out by the 'circuit data errors diagnostic phase' of the program.

(b) Instructions for the user

(i) Convert the circuit into its equivalent tabular representation by the technique discussed in Section 3 of the Chapter.

(ii) Ensure that the number of nodes in the circuit does not exceed 500, and that the circuit is purely combinatorial comprising only of NAND/NOR/AND/OR/X-OR/NOT gates.

(iii) Specify SIMTYP = 1 for only 'True Value Simulation', and SIMTYP = 2 for fault simulation.

(iv) Total number of gates (including Primary inputs) is indicated by 'N'.

(v) No. of test-points and the serial numbers of nodes to be specified as test points are indicated respectively by 'NTPTS' and 'TESTPT(KZ)'.

(vi) No. of user specified test vectors is termed as 'NTEST'. If no such vector is specified, the data card must read '0'.

(vii) Test vectors can be specified in 3 ways - viz.

MANUAL=0, means no test vector is supplied by the user (i.e., all these should be generated randomly).

=1, means user supplied test vector. In this case all primary input values have to be specified by the user.

=2, means the test vector is partially supplied by the user and partially generated randomly. In this case the S.No. of the primary input which is desired to carry the value '1' should be indicated by the user. Other inputs are randomly generated. Note that the user cannot specify a particular input = 0.

To specify these inputs the variable 'NSET' has been used in the program. In case the **user wants to service only the test vectors** specified by him and no other randomly generated test vector then he should put 'NEND value in Statement No. 2543 of the program equal to 'NTEST'. In general 'NEND' value (i.e., No. of test vectors) to be serviced has been specified as '$\{(2)^{IPS} + 10\}$' where IPS is the number of primary inputs in the circuit.

(viii) No. of primary output points is indicated as 'KTM' and such nodes are specified as 'PRIOUT(JTM)'.

(ix) Various data cards required are to be arranged in the following sequence and punched as per the format used in the program statement numbers shown against each.

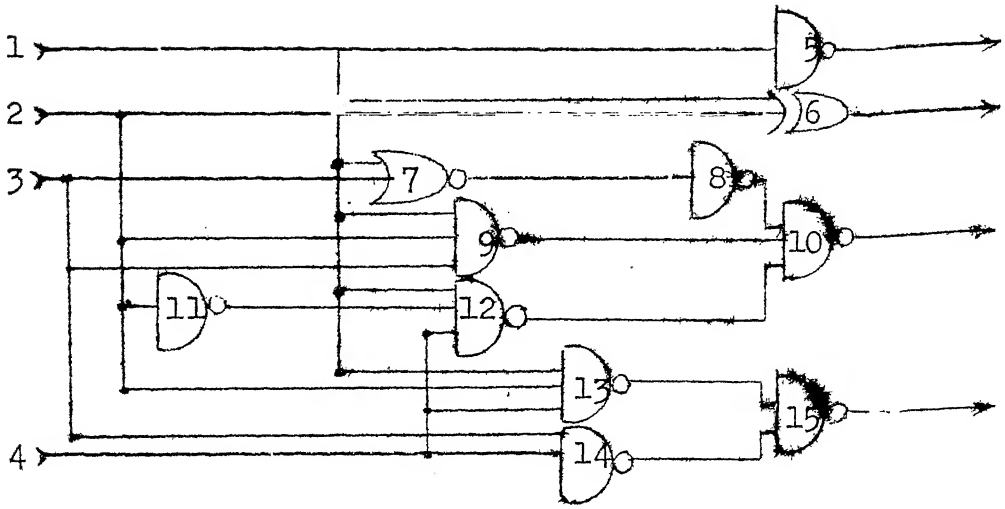
S.No.	Data	Statement No. in the Program
1	N, SIMTYP	10
2	NTPTS, TESTPT	12
3	NTEST	14
4	Circuit data consisting of: GATENO, INPUTS, COMP, FANOUT, GLINK	20
5	KTM	51066
6	PRIOUT	51067
7	MANUAL, NSET	2602

Note: If NTEST=0, the last data card need not be fed at all since the program branches to generate the test vectors randomly.

Example

For the 'Excess-3' to BCD Converter' circuit shown below write the data cards assuming the following:

(i) Fault simulation is required.
(ii) No test points are specified by the user.
(iii) Only test vector specified is '1111'.
(iv) No additional primary output point is specified.

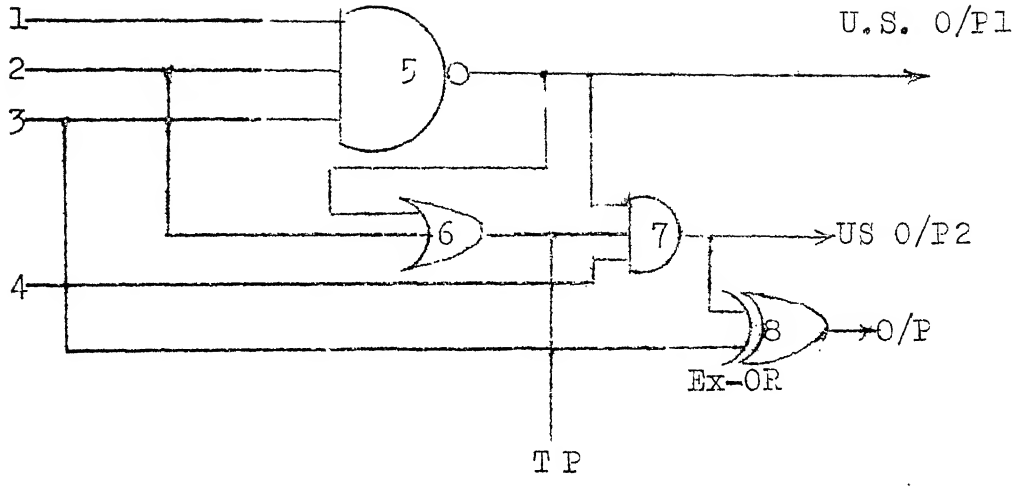


Data Cards - (Reader to verify corrections!)

S.No. of data card	Data punched starting with column No. 1
1	ßßß15ß2
2	ß0ßß0
3	ß1
4	ßß1-106ßßß5.2ßßß6.2ßßß7.2ßßß9.2ßß12.2 ßß13.21000.
5	ßß2-104ßßß6.3ßßß9.3ßß11.2ßß13.31000.
6	ßß3-103ßßß7.3ßßß9.4ßß14.21000.
7	ßß4-103ßß12.4ßß13.4ßß14.31000.
8	ßß5ß2101000.
9	ßß6ß6 201000.
10	ßß7ß421ßßß8.21000.
11	ßß8ß211ßß10.21000.
12	ßß9ß231ßß10.31000.
13	ß10ß2301000.
14	ß11ß211ßß12.31000.
15	ß12ß231ßß10.41000.
16	ß13ß231ßß15.21000.
17	ß14ß221ßß15.31000.
18	ß15ß2201000.
19	ß0
20	ßß0
21	1ßß1ß1ß1ß1

Total number of data cards = 21.

Note: 'ß' means blank.



(Back Reference : Page No. 59)

Note: US O/P means user specified output point

TP means test point.

CHAPTER 4

F U T U R E D E V E L O P M E N T A L S C O P E

CHAPTER 4

FUTURE DEVELOPMENTAL SCOPE

1. FAULT SIMULATION ON SEQUENTIAL CIRCUITS

Generation of fault-lists for 'flip-flops' is difficult compared to the gates since a FF can 'remember' the effects of faults which were propagated to its inputs at a previous time in a test sequence. For example, at some time in the test sequence a fault 'f' might cause a FF to be erroneously set. The erroneous setting signal might then disappear, but in the absence of a subsequent re-setting signal, the FF will remain set, and the fault is detectable at its output so long as the erroneous set state persists.

To correctly account for this 'memory' property of FF's, the new fault lists computed at time 't_i' should include:

(a) faults appearing in the fault lists on the input terminals at time 't_i'.

(b) faults contained in the fault-lists at output terminals at the previous time interval 't_{i-1}'.

(c) internal faults of the FF detectable at the output at time 't_i'.

Sample Computation of Fault-Lists for a NOR Latch

Consider the FF structure and the associated fault lists at output points P_1 and Q_1 at time t_1, P_2 and Q_2 at time t_2, and the input fault lists S and R at time t_2

as shown in the figure below:

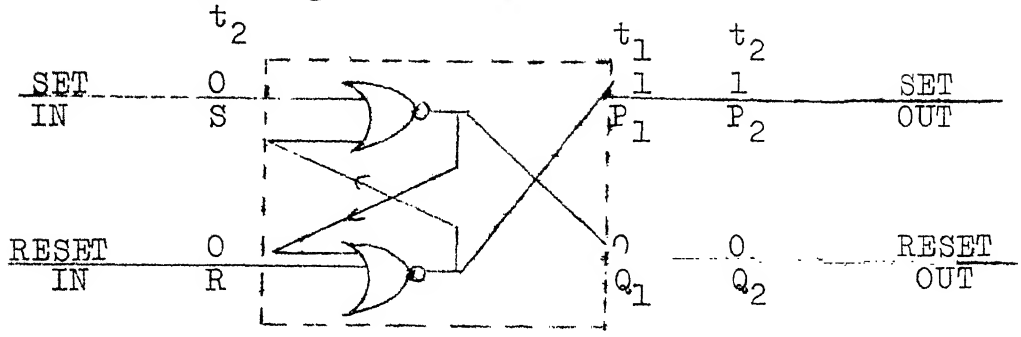


Figure 1

It may be observed that the FF is 'set' at time 't_1', and at time 't_2' the input values at SET, RESET terminals both change to '0'. We are to evaluate the fault-lists P_2, Q_2 at time t_2. It is apparent that the FF remains 'set' at 't_2' since 'non-enabling' inputs have been applied.

Since P_2, Q_2 are to depend on the 16 fundamental products of P_1, Q_1, R and S (viz., $\bar{P}_1\bar{Q}_1\bar{R}\bar{S}$, $\bar{P}_1\bar{Q}_1\bar{R}S$, ..., P_1Q_1RS), we should examine each one of these combinations separately to see if it should be present in the expression for P_2 or Q_2.

To determine whether the product P_1Q_1RS should be present in P_2 or Q_2, let us consider its meaning. Any fault in the product P_1Q_1RS, if actually present would, by definition, cause both outputs to be wrong at t_1, and both inputs to be wrong at 't_2'. That is, it will cause the FF to be in the reset state at 't_1' and would make both inputs '1' (i.e., both enabled) at time t_2. This

causes both FF outputs to go to '0' at t_2, whereas the 'good' FF state at 't_2' is 'set'. Therefore this fault (i.e., product P_1Q_1RS) can be detected on the 'set' output at t_2, but not on the 'reset' output. Consequently, the product P_1Q_1RS is present in the expression for P_2, but is absent from the expression for Q_2.

To further clarify, consider the product $P_1Q_1R\bar{S}$. Any fault in this set would cause the FF to be in the reset state at time t_1, and would cause 'reset enable' signal to be applied at the inputs at time t_2 which would maintain the FF in the reset state. Therefore, this fault would be detected at both outputs at t_2. So $P_1Q_1R\bar{S}$ will appear in the expressions for both P_2 and Q_2.

(Note: Meaning of $P_1Q_1R\bar{S}$ is that except for 'S' which is non-effective, P_1, Q_1, R all tend to invert the true value at that point.)

The reasoning for the remaining 14 products is in a similar vein as above although in some cases special considerations apply as follows:

(a)Race Condition: We consider the product $P_1\bar{Q}_1\bar{R}\bar{S}$. Any fault in this would cause both outputs of the FF to be '0' at 't_1' implying that both inputs must be '1' at that time. Moreover, the fault $P_1\bar{Q}_1\bar{R}\bar{S}$ does not change the status of inputs at 't_2' which remain as '0'. This means that, with the fault present, both inputs will change from '1' to '0' in the interval (t_1, t_2), and it is

not known which input changes first. Consequently it is unpredictable if the FF at 't_2' will end up in the 'set' or 'reset' state. This is the so-called 'race-condition'.

Such faults which are not truly detectable at the output are termed as 'STAR FAULTS'. Simulator, while evaluating fault lists on the FF outputs, should include the star faults along with the regular faults, but should distinguish these with a 'flag' on the star fault entry.

(b) Don't Care Products: There is no fault external to a (cross-coupled NOR) FF which can cause both FF outputs to be '1' simultaneously. For the particular FF state shown in the Figure '1', such faults can only appear in the 4 products containing $\bar{P}_1 Q_1$. Since we are considering faults only external to the FF at this stage, we may, therefore, treat these four products as 'Don't Care's', and include them in the expression for P_2 and Q_2 if their inclusion permits simplication of the resulting expression.

An analysis for all fundamental products for this example results in the following sum-of-products expressions for P_2 and Q_2:

$$P_2 = \{\bar{P}_1\bar{Q}_1R\ \bar{S}\} + \{P_1Q_1RS + P_1Q_1R\bar{S} + P_1Q_1\bar{R}\ \bar{S} + P_1\bar{Q}_1R\ S + P_1\bar{Q}_1R\ \bar{S} + \bar{P}_1\bar{Q}_1R\ S + \bar{P}_1\bar{Q}_1R\ \bar{S} + (\bar{P}_1Q_1R\ S + \bar{P}_1Q_1R\ \bar{S} + \bar{P}_1Q_1R\ \bar{S})\}$$

$$Q_2 = \{P_1\bar{Q}_1R\ \bar{S}\} + \{P_1Q_1R\ \bar{S} + P_1Q_1\bar{R}\ \bar{S} + P_1\bar{Q}_1R\ \bar{S} + \bar{P}_1\bar{Q}_1R\ S + (\bar{P}_1Q_1R\ \bar{S} + \bar{P}_1Q_1R\ \bar{S})\},$$

where the first bracketed products represent the 'star faults' and the products in the paranthesis are 'don't Care's'.

After simplification of the 'non-star' faults we obtain:

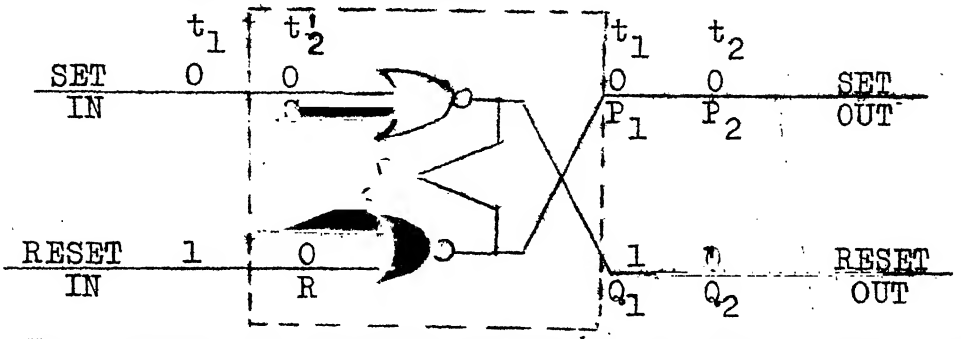
$$P_2 = \{\bar{P}_1\bar{Q}_1R\ \bar{S}\} + R + Q_1\bar{S} \qquad \text{(i)}$$

$$Q_2 = \{P_1\bar{Q}_1R\ \bar{S}\} + R\ \bar{S} + Q_1\bar{S} \qquad \text{(ii)}$$

Expressions (i) and (ii) for P_2 and Q_2 are incomplete without the internal FF faults detectable at 't_2'. For example, internal faults are two, labelled 'A_s' and 'A_r', which hold the FF in the set and reset state respectively regardless of the inputs. In this example, the fault 'A_r' would be detectable on both the set and reset outputs at t_2 and would, therefore, be included in both P_2 and Q_2, whereas 'A_s' would not be detectable at t_2 and will not appear in either P_2 or Q_2.

Example

Make a tabular representation of detectable faults at P_2 and Q_2 for the input/output conditions of the NOR latch shown below:



If a fault is included in P_2/Q_2 we shall represent it as '1', otherwise as '0'.

S. No	FAULT				FAULT PROPAGATION STATUS AT	
	P_1	Q_1	R	S	P_2	Q_2
1	0	0	0	0	0	0
2	0	0	0	1	1	1
3	0	0	1	0	0	0
4	0	0	1	1	0	1
5	0	1	0	0	'Race Condition'-Star Fault	
6	0	1	0	1	1	1
7	0	1	1	0	0	0
8	0	1	1	1	0	1
9	1	0	0	0	'Don't Care Condition'	
10	1	0	0	1	'Don't Care Condition'	
11	1	0	1	0	'Don't Care Condition'	
12	1	0	1	1	'Dont Care Condition'	
13	1	1	0	0	1	1
14	1	1	0	1	1	1
15	1	1	1	0	0	0
16	1	1	1	1	0	1

Minimizing the expressions with Karnaugh Maps using Don't Care Conditions (represented by ϕ).

RS \ P_1Q_1	00	01	11	10
00	0	0	1	ø
01	1	1	1	ø
11	0	0	0	ø
10	0	0	0	ø

RS \ P_1Q_1	00	01	11	10
00	0	0	1	ø
01	1	1	1	ø
11	1	1	1	ø
10	0	0	0	ø

We get, $P_2 = P_1\bar{R} + \bar{R}S$ and $Q_2 = S + P_1\bar{R}$

Including the star-faults and the internal faults of FF 'A's and 'A_r' we get the expressions for P_2 and Q_2 as

$$P_2 = \{\bar{P}_1 Q_1 \bar{R}\ \bar{S}\} + P_1\bar{R} + \bar{R}\,S + A_s$$

$$Q_2 = \{\bar{P}_1 Q_1 \bar{R}\ \bar{S}\} + S + P_1\bar{R} + A_s.$$

We have also worked out the fault lists for JK flip-flops assuming that the 'clock' input is not faulty. The results for various input combinations attempted are tabulated below which may be verified by the reader:

S. No.	Time	INPUT COMBINATION J	K	OUTPUT		FAULT LIST AT t_2
1	t_1	1	0	1	0	$P_2 = P_1 + K$
	t_2	1	1	0	1	$Q_2 = P_2 + Q_1\bar{J}$
2	t_1	1	1	1	1	$P_2 = P_1 + K$
	t_2	0	1	0	1	$Q_2 = P_2 + Q_1 J$
3	t_1	0	1	0	1	$P_2 = Q_2 + P_1\bar{K}$
	t_2	1	1	1	0	$Q_2 = Q_1 + J$
4	t_1	0	1	0	1	$P_2 = Q_2(K+\bar{P}_1)$
	t_2	0	0	0	1	$Q_2 = \bar{Q}_1 J$

Likewise for different other input combinations of the FF's similar expressions can be worked out for P_2 and Q_2 respectively. But this process, we should be able to express in the form of a general algorithm applicable to all types of FF's. This generalization can be included in the future development program of this deductive technique.

Circuits with Feedback Loops: We have already catered in our simulator for the sensing and solution of oscillatory conditions and are able to simulate combinational logic circuits with global feedback. Ignoring the races and reducing the FF to its equivalent gate level model, we have also been able to simulate a 'SR' FF with the simulator already designed by us. For this we decided to specify ourselves the test vectors to be simulated, rather than generating these randomly, and deliberately avoided the simultaneous changing of both R and S inputs from '0' to '1' which results in a race-condition for the RS FF made of cross-connected NAND gates. This input combination change from (0,0) to (1,1) should be 'self-inhibited' for a SR FF if the simulator is designed to cater for all types of sequential circuits.

With the addition of 'STAR FAULTS' and by changing the 'gate-level' model into an 'element level' model, this simulator can be made more universal in its applications. Further modification may be attempted by placing the FF's on the feedback loops. Then the fault computation for more 'complex logic devices' like 'shift registers', 'counters' etc., could also be included. All these additions/modifications in the program enhance the scope of future development on the work already undertaken.

2. ADDITIONAL FAULTS POSSIBLE

We, in our simulation model, have considered only the 'stuck-at' type of faults. In practice, however, various other faults are also encountered in logic circuits, the detection of which should be feasible and may be taken up by those interested in further probing into this deductive technique of fault simulation. Some of the faults to be considered are as under:

(a) Delay-Faults: These are due to incorrect delay values associated with various gates which cause the circuit behaviour to be different from what was intended. Mostly these are due to the complexity of manufacturing processes. (Refer 13).

(b) Shorted Faults: These are the non-classical faults like 'cross-over shorts' and 'shorts between adjacent paths'. These normally would be occuring during the manufacturing processes for IC's. For such type of faults a separate simulator subroutine may be developed which could be called when the design of 'manufacturing tests for circuit pack check out' is required.

3. VARIABLE TIME DELAY SIMULATION

We had selected the unit time delay model since it does not require storage of multiple-time-frame evaluations, and can work with a time-queue for the next step only. But this model does not truely represent the 'delays' associated with different types of gates. It should be possible to assign different delays to different gates and then simulate the circuit behaviour. The complexity of process involved calls for increased storage requirements and extra time for simulation, but will certainly make the simulation more accurate. It shall be worthwhile attempting a variable-time delay model using the deductive technique.

CONCLUSION

In the preceding chapters we have attempted to develop a fault simulator for logic circuits by the DEDUCTIVE Approach. Our emphasis throughout has been on 'memory space' considerations of the host computer (IBM-7044) and on the 'simulation simplicity'. Since our aim was merely the implementation of the 'Deductive technique', we modelled only the 'stuck-at' type faults with 'unit time-delay' associated with each logic gate.

It goes without saying that to make the scope of application of this simulator more diverse, a good deal of work is yet to be done. Modelling with 'variable/assignable-time-delay', and inclusion of 'element-level subroutines' shall be a desirable extension of the work already undertaken. The simulator, then, can be an effective tool in the hands of both the logic designer and the maintenance engineer.

REFERENCES

1. 'Fault Detection in Digital Circuits' by Friedman and Menon.

2. 'The Diagnosis of Asynchronous Sequential Switching Systems', Seshu, S., and Freeman, D.N. (IRE Transactions of Electronic Computers, Vol. EC-11, August, 1962.)

3. 'A Deductive method for Simulating Faults in Logic Circuits', Armstrong, D.B., (IEEE Transactions on Computers, Vol. C-21, May 1972)

4. 'Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits', by J.P. Roth, W.G. Bouricin and P.R. Schneider, (IEEE Transactions on Computers, Vol. EC-16, October 1967.)

5. 'On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets', D.B. Armstrong, (IEEE Transactions on Electronic Computers, Feb. 1966.)

6. 'Design Automation of Digital Systems', M.A. Breuer, (Vol. 1, Chapter 3).

7. 'Automatic Digital Test Generation for Sequential and Combinational Logic', by Melvin A. Breuer, (Published in the 1972 Proceedings of the National Electronics Packaging Conference (NEPCON), Anaheim, California.)

8. 'Fault Diagnosis of Digital Systems', H.Y. Chang, E.G. Manning and G. Meteze.

9. 'Digital Logic Simulation in a Time-Based, Table-Driven Environment', by S.A. Szygenda and E.W. Thompson, (IEEE Computer Society Journal - 'COMPUTER' - March 1975.)

10. 'Deductive Techniques for Simulating Logic Circuits', H.Y. Chang and S.G. Chappell ('COMPUTER' - March 1975).

11. 'LAMP : Logic Analyzer for Maintenance Planning', (The Bell System Technical Journal, Vol. 53, No. 8, October 1974.)

12. 'A Variable Time-Delay Subroutine for Digital Simulation, Programs', Ball, J.S. (Simulation, Vol. 7, No. 1, July, 1966).

13. 'The Effect of Races, Delays and Delay Faults on Test Generation', Melvin A. Breuer (yet to be published).

A46796

Date Slip

This book is to be returned on the date last stamped.

.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....

CD 6.72.9

EE-1976-M-BHA-DED